UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

novaVM: Enhanced Java Virtual Machine for Big Data Applications

Rodrigo Fraga Barcelos Paulus Bruno

Supervisor: Doctor Paulo Jorge Pires Ferreira

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction and Honour

2018

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

novaVM: Enhanced Java Virtual Machine for Big Data Applications

Rodrigo Fraga Barcelos Paulus Bruno

Supervisor: Doctor Paulo Jorge Pires Ferreira

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction and Honour

Jury

Chairperson: Doctor Rodrigo Seromenho Miragaia Rodrigues, Instituto Superior Técnico, Universidade de Lisboa

Members of the Committee:

Doctor Christoph Kirsch, Faculty of Natural Sciences, University of Salzburg, Áustria

Doctor Paulo Jorge Pires Ferreira, Instituto Superior Técnico, Universidade de Lisboa

Doctor Bruno Miguel Brás Cabral, Faculdade de Ciências e Tecnologia, Universidade de Coimbra

Doctor António Paulo Teles de Menezes Correia Leitão, Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions:

Fundação para a Ciência e Tecnologia

2018

Acknowledgments

Before presenting the work that was developed during the course of this PhD program, I would like to leave a few words of gratitude to those that really helped, stood by my side, and made my journey much more joyful.

First, I need to deeply thank my advisor, Professor Paulo Ferreira, who started mentoring me even before my master thesis. Professor Paulo always believed in me and was also very supportive and ready to help, presenting me with challenges that made me pursue higher goals. I feel honored for having the opportunity to learn so much from him.

To my close family, my mother Ana, father Jorge, sister Catarina and Daniel, who always stood by my side and provided me with all the possible support. I greatly appreciate your understanding even when I had to work and could not spend more time with you, specially during vacation.

Finally, to Cláudia, my long life partner that always motivated me even in the most difficult moments, helping me go through. I cannot express enough gratitude for the fact that you supported me during long nights of work and long travels.

To finish, I also need to thank FCT (Fundação para a Ciência e Tecnologia), the entity that gracefully supported my PhD program through the grant number SFRH/BD/103745/2014.

Resumo

A necessidade de processar grandes quantidades de dados, i.e., Big Data, é uma realidade atual. Aliadas a esta necessidade e apresentando ciclos de desenvolvimento rápidos e grande quantidade de recursos desenvolvidos pela comunidade, linguagens recentes tais como o Java, Scala e Python tornaram-se as linguagens de preferência para escrever aplicações Big Data. No entanto, as aplicações escritas nestas linguagens são comummente executadas, utilizando um sistema de gestão de execução que não toma em consideração as necessidades de uma aplicação Big Data. Em particular, no contexto deste trabalho, identificamos três problemas: i) a necessidade de recuperar falhas ou criar novas réplicas rapidamente; ii) a necessidade de melhorar a gestão de memória usada no sistema de gestão de execução, por forma a permitir aumentar a quantidade de memória usada sem comprometer os tempos de latência da aplicação; iii) a necessidade de gerir recursos eficientemente e evitar o desperdício de recursos. Estes problemas são fundamentais para a execução de aplicações Big Data e não podem ser resolvidos, usando soluções existentes.

Para resolver os problemas acima apresentados, o presente trabalho propõe um conjunto de algoritmos: i) ALMA, um algoritmo de migração/replicação que tira partido de informação interna de gestão de memória para melhorar a eficiência do processo de migração/replicação de aplicações; ii) NG2C, um algoritmo de gestão de memória com múltiplas gerações que permite reduzir os tempos de latência das aplicações; iii) POLM2 e iv) ROLP, dois algoritmos que captam, ainda que diferente forma, informação sobre a alocação de objetos da aplicação e que, pode ser utilizada para configurar o algoritmo NG2C; v) Escalabilidade Vertical Dinâmica, um algoritmo que propõe uma nova abordagem de redimensionamento da memória que permite melhorar a gestão da mesma e reduzir desperdícios.

Os algoritmos propostos estão implementados como sub-componentes da novaVM, uma Máquina Virtual Java (JVM), implementada tendo como base a OpenJDK 8 HotSpot JVM, uma JVM vastamente utilizada na indústria. Para cada algoritmo, apresentamos ainda uma série de experiências utilizando uma combinação de aplicações reais e sintéticas. Os resultados são muito promissores, confirmando a utilidade das contribuições propostas.

A investigação e os trabalhos aqui apresentados são suportados por uma coleção de artigos publicados em revistas e conferências internacionais. Além disso, foram também realizadas várias contribuições de código fonte desenvolvido no intuito deste trabalho para projetos opensource de grande relevância tais como o CRIU e a OpenJDK.

Palavras Chave: Big Data, Máquina Virtual Java, Reciclagem de Memória, Gestão de Memória, Latência.

Abstract

The need to process large amounts of data, i.e. Big Data, is a reality. From scientific experiments to social networks, Big Data applications require processing and storing massive amounts of data in an efficient way. In addition, with fast development cycles and large community resources, managed programming languages such as Java, Scala, and Python are now the preferred languages to implement Big Data applications.

However, these languages run on top of managed runtimes that were not built to cope with the challenges imposed by Big Data applications. In particular, this work identifies three problems/challenges that need to be addressed: i) the need to quickly recover from failed nodes or to spawn more nodes to accommodate new workload demands; ii) the need to improve runtime memory management to be able to scale to large amounts of data in memory without sacrificing the application latency; iii) the need to efficiently manage resources and minimize resource waste. These are fundamental problems to most Big Data applications running on managed runtimes and can not be solved using previously proposed solutions.

To solve the aforementioned problems, this work proposes a number of algorithms: i) ALMA, a migration/replication algorithm that takes advantage of internal memory management information to improve the runtime migration/replication; ii) NG2C, an N-Generational Garbage Collector that reduces applications' long tail latencies; iii) POLM2, an offline profiler that can be used to profile workloads, and whose output information can be used to configure NG2C; iv) ROLP, an online profiler, running inside the runtime that automatically profiles the application and configures NG2C; v) Dynamic Vertical Scaling, a new heap sizing strategy that improves runtime resource management to reduce resource waste.

All the proposed algorithms are implemented as sub-systems of novaVM, a new Java Virtual Machine (JVM), implemented on top of OpenJDK 8 HotSpot JVM, a widely-used industrial JVM. Each algorithm is evaluated using benchmarks and workloads based on real-world applications. Results are very promising, demonstrating that the proposed goals were achieved.

This work is supported by a number of publications in international journals and conferences. In addition, novaVM's source code is opensource and part of it is now included in several opensource projects such as CRIU and OpenJDK.

Keywords: Big Data, Java Virtual Machine, Garbage Collection, Memory Management, Latency.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

We start this document by presenting an overview of the work described in the next chapters. At first, we provide motivation and explain why it is relevant to pursue this research area. Then, precisely describe the goals and requirements for the solutions that we discuss shortly after. This chapter closes with a list of research and code contributions that resulted from this work.

1.1 Big Data and Runtime Systems

The need to efficiently process large amounts of data, Big Data, to extract valuable information, is a reality [1]. Scientific experiments (such as protein folding, physics simulators, and signal processing), social networks, fraud detection, and financial analysis are just a few examples of areas in which large amounts (thousands or even hundreds of thousands of GBs) of information are generated and processed daily. Moreover, the importance of live and strategic information has led the biggest companies in the world (Google, Facebook, Oracle, Yahoo!, Tweeter, Amazon, among others) to build Big Data platforms.

Big Data platforms are designed to efficiently handle massive amounts of data. There are many examples of such platforms: Hadoop [123] (a MapReduce implementation), Cassandra [79] (a distributed Key-Value store), Neo4J [104] (a graph database), GraphChi [78] (a graph engine), Spark [128] (a cluster computing system for Big Data applications), Google File System [49] (a distributed file system), Naiad [93] (a dataflow system), Dryad [63] (a scheduler for distributed Big Data applications), Lucene [90] (an in-memory indexing tool), etc.

It is also a fact that many of these Big Data platforms are written in managed languages such as Java, Scala, and Python, among others. These languages are often the developer's choice for implementing such platforms due to its quick development cycle and rich community resource. To support the execution of these languages, runtime systems such as the Java

Virtual Machine (JVM) are used. Given the importance of the these platforms and the runtimes used to support them, this work focuses on studying the scalability issues of runtime systems (the JVM in particular) imposed by Big Data platforms.

There are numerous real-world examples of Big Data applications running on top of a runtime system. We now present some of those examples, in particular those that we had the opportunity to closely interact with. First, Feedzai's[1] credit card transaction validation service is a system that uses information stored in JVM-powered databases whose access time is crucial for Feedzai. Failing a Service Level Agreement (SLA) for a particular transaction due to a long database read or write operation leads to a significant negative impact for the company. Second, Jelastic is a JVM centered cloud provider[2] that handles thousands of customer instances running JVM applications that need to be managed in an efficient way. For Jelastic, it is imperative to be able to migrate or replicate instances through physical nodes and to be able to dynamically adapt the resources given to each instance.

1.2 Goals

Driven by the scalability needs imposed by Big Data platforms, our work focus on three main problems:

- Problem 1 (horizontal scalability): the need to quickly recover from failed nodes or to spawn more nodes to accommodate new workload demands;
- Problem 2 (long tail latency): the need to improve runtime memory management to be able to scale to large amounts of data in memory without sacrificing application performance in terms of latency;
- Problem 3 (vertical scalability): the need to efficiently manage resources and to adapt the runtime's resource usage according to the application needs to avoid both application performance problems due to lack of resources and, at the same time, resource waste.

In addition to solving the aforementioned problems (horizontal scalability, long tail latency, and vertical scalability), this work strives to provide a unified JVM that solves these problems while respecting the following requirements:

[1] Feedzai (www.feedzai.com) is a world leader data science company that detects fraud in omnichannel commerce. The company uses near real-time machine learning to analyze Big Data to identify fraudulent payment transactions and minimize risk in the financial industry.

[2] Jelastic is a decentralized multi-cloud provider that introduced the "pay-as-you-use" model, a model where customers are billed based on used resources and not reserved. It can be reached at jelastic.com

- Requirement 1: minimal application throughput overhead with regards to the application normal execution;
- Requirement 2: minimal application memory footprint increase with regards to the application normal memory footprint;
- Requirement 3: zero programmer effort, i.e., the programmer should not have to change the code, or the environments where the JVM runs in order take advantage of novaVM. In other words, novaVM must work as a drop-in replacement for current JVMs.

It is important to note that none of the previously presented problems can be solved using current JVMs because they are not architecturally designed to cope with the goals and problems just mentioned (which are important for a wide spectrum of Big Data applications). Moreover, the memory management problem is specially challenging in managed programming languages. While the use of such languages makes programming easier, their automated memory management (Garbage Collection) comes at a cost which greatly limits the scalability of an application [102]. This cost is significantly magnified when these managed runtimes are used to run Big Data applications, which tend to harvest computational resources.

In addition, numerous solutions have been proposed to solve each of the presented problems. However, no previous solution succeeded in solving these problems while still ensuring low throughput overhead, low memory footprint impact, and no programmer effort, all at the same time. Chapter 3 presents further discussion on related works showing a detailed study of current solutions for each problem and explaining why these solutions do not meet our goals.

Note that, switching back to an unmanaged language such as C or C++ is not a reasonable choice for several reasons. First, unmanaged languages are error-prone and debugging memory problems is known to be a long and painful task. Second, since a great number of existing Big Data platforms are already developed in a managed language, it is unrealistic to re-implement them from scratch. In addition, in many applications and platforms, the bottleneck for improving performance is I/O (i.e., input and output operations). For such scenarios, switching to unmanaged languages has zero benefit.

1.3 Proposed Solution

To solve the aforementioned problems and to improve the runtime system support for Big Data applications, we present novaVM, an enhanced Java Virtual Machine (JVM) for Big Data applications. The proposed solution, novaVM, achieves the goals envisioned for this work through the following developed algorithms, implemented as sub-systems of novaVM:

- ALMA - this sub-system provides fast JVM replication/migration by being able to snapshot a local JVM and migrate it to a remote node (from now on, we will refer to this process as only migrating the JVM). ALMA reduces both the total migration time (time between the migration starts and the new JVM starts) and the pause time (time between the old JVM stops and the new one starts) by minimizing the number of objects included in the JVM snapshot. This is achieved with the help of the garbage collector by discarding unreachable objects i.e., only the live working set is included in the snapshot. More details on Section 4.2;

- NG2C - this sub-system represents a new approach to generational collectors. NG2C provides a new heap layout, with an arbitrary number of generations as opposed to current generational collectors which have a limited number of generations, usually two. By using N generations, NG2C is able to reduce object copying and heap fragmentation since objects with different life time estimates are allocated in different generations. Compared to current generational collectors, NG2C is able to reduce application pause times as fewer objects need to be copied during garbage collections. NG2C supports object allocation annotations to specify in which generation to allocate an object. More details on Section 4.3;

- POLM2 - this sub-system is an offline profiler that produces object life times estimates by tracking object allocation and collection. These estimates are used to automatically instrument application Bytecode to take advantage of NG2C with no programmer effort. POLM2 greatly reduces the complexity of instrumenting the code of the application, necessary to use NG2C. More details on Section 4.4;

- ROLP - this sub-system is an online profiler that produces object life time estimates by internally tracking object allocation and collection. In other words, this component is working inside the JVM (as opposed to POLM2 which monitors the JVM from the outside). ROLP dynamically adapts decisions to allocate objects in different generations according to the application allocation behavior. As opposed to POLM2, no previous application workload knowledge is required to take advantage of NG2C. Note that this also means that this solution (opposed to POLM2) also dynamically adapts to changing workloads. This comes at a negligible throughput cost. More details in Section 4.5;

- Dynamic Vertical Scaling - this sub-system provides a new heap sizing approach to improve resource efficiency in vertical scaling. This new heap sizing technique forces the JVM to adapt its resource usage according to the application needs. This is specially

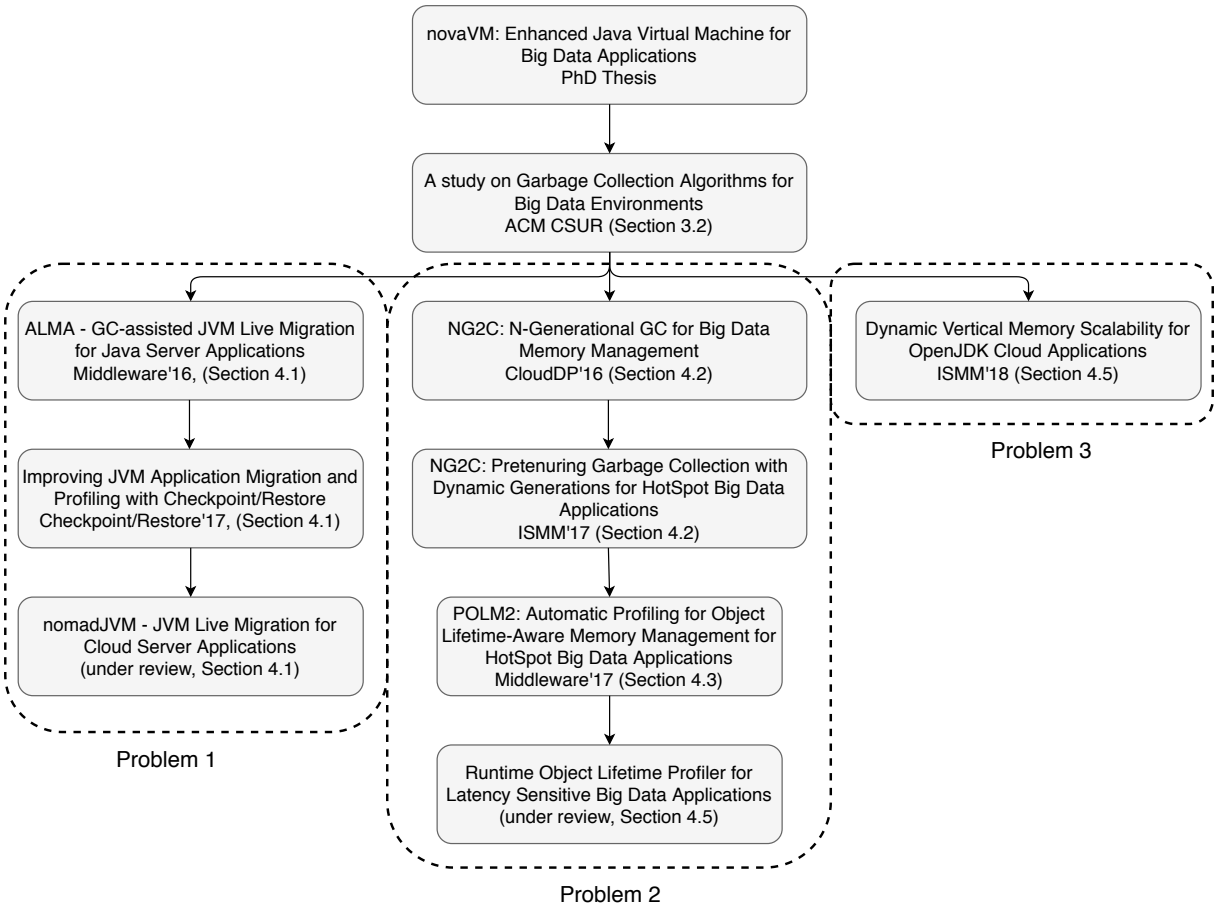

Figure 1.1: novaVM Research Path

important for cloud providers that are shifting the billing model from forcing the user to pay for what was reserved into letting the user pay for the amount of resources that are actually being used. More details in Section 4.6.

Figure 1.1 shows a graphical representation of how the research work in this thesis is organized, including the list of corresponding publications (by conference or journal acronym and year), as well as the section where the respective work is presented.

Starting from the top, the first component in novaVM is a study on current Garbage Collection algorithms and how these algorithms affect the performance of applications in Big Data environments. From this study, three main problems are identified (as described in Section 1.2) where GC could be used and improved to provide a better solution compared to current alternatives. Hence, three new algorithms (implemented as sub-systems in novaVM) are proposed: ALMA, NG2C, and Dynamic Vertical Scaling. Each algorithm solves one of the proposed problems: ALMA is a JVM replication/migration tool that provides solution to Problem 1; NG2C is an N-Generational GC that solves Problem 2; Dynamic Vertical Scaling enables the JVM to better manage resources, solving Problem 3. As NG2C requires programmer effort and knowledge to

annotate application code, POLM2 and ROLP, are designed to satisfy the requirement related to having no need for programmer effort. Note that POLM2 and ROLP provide two different alternatives to replace programmer effort and knowledge, with different granularity control and performance trade-offs (this topic is further discussed in Chapters 4 and 6). It is important to note that all proposed algorithms satisfy the other two requirements: minimal overhead on application throughput and memory footprint.

1.4 Contributions

In the following, we summarize the main contributions of this work:

- design and implementation of ALMA, NG2C, Dynamic Vertical Scaling, POLM2, and ROLP, the main algorithms that are included in novaVM;
- evaluation of the proposed algorithms using a combination of real-world and synthetic applications, which show significant performance improvements with regards to current approaches;
- a comprehensive analysis of Big Data environments' memory profiles, i.e., how objects created by Big Data applications are kept in memory and which challenges arise from these applications;
- an analysis of recent and significant migration algorithms, mostly targeted to applications with tight requirements regarding throughput and latency (such as Big Data platforms);
- an analysis of recent and significant GC algorithms targeted to Big Data platforms, which deal with the main scalability challenges inherent in Big Data environments: throughput and latency;
- an analysis of current resource management approaches and challenges of Big Data platforms in the cloud.

In addition to the aforementioned contributions, all the developed code is opensource and is available at `github.com/rodrigo-bruno`. Finally, we would like to emphasize two contributions to the opensource community:

- a patch was proposed and accepted to integrate part of the ALMA code into CRIU [71]. The developed code for disk-less network migration is now part of this Linux Checkpoint/Restore tool, used in projects such as Docker;

- two patches were proposed and accepted to integrate the Dynamic Vertical Scaling code in OpenJDK HotSpot JVM. The functionality introduced by the patches is expected to be available in OpenJDK 12.

To finish, a set of research works were also accepted in major international journals and conferences (ranked as A* or A by CORE):

- **NG2C: N-Generational GC for Big Data Memory Management**. Rodrigo Bruno and Paulo Ferreira. In *6th International Workshop on Cloud Data and Platforms* (co-located with EuroSys). 2016, London, United Kingdom;
- **ALMA - GC-assisted JVM Live Migration for Java Server Applications**. Rodrigo Bruno and Paulo Ferreira. In *Proceedings of the 17th Annual Middleware Conference*. 2016, Trento, Italy [22];
- **NG2C: N-Generational GC for Big Data Applications**. Rodrigo Bruno, Luís Oliveira, and Paulo Ferreira. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 2017, Barcelona, Spain [23];
- **Improving JVM Application Migration and Profiling with Checkpoint/Restore**. Rodrigo Bruno and Paulo Ferreira. In *International Microconference on Checkpoint/Restore* (co-located with Linux Plumbers). 2017, Los Angeles, USA;
- **POLM2: Automatic Profiling for Object Life Time-Aware Memory Management for Hotspot Big Data Applications**. Rodrigo Bruno and Paulo Ferreira. In *Proceedings of the 18th Annual Middleware Conference*. 2017, Las Vegas, USA [21];
- **A study on Garbage Collection Algorithms for Big Data Environments**. Rodrigo Bruno and Paulo Ferreira. In *ACM Computing Surveys*. 2018 [22];
- **Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications**. Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, Philadelphia, PA, USA [24];
- **ROLP: Runtime Object Life Time Profiling for Big Data Memory Management**. Rodrigo Bruno, Duarte Patrício, José Simão, Luís Veiga, and Paulo Ferreira. Currently under review for an international conference;
- **nomadJVM - JVM Live Migration for Cloud Server Applications**. Rodrigo Bruno and Paulo Ferreira. Currently under review for an international journal;

1.5 Road Map

We begin our work with a comprehensive study of Big Data environments (Section 2.1), which is divided in the analysis of processing and storage platforms (Sections 2.1.1 and 2.1.2, respectively). Before delving into how memory management works, Section 2.2 discusses important internal JVM architectural concepts and mechanisms important to understand how the JVM works. Then, we provide a complete background on memory management (Section 2.3) and classic GC algorithms (Section 2.4), both important topics to understand the design of novaVM.

Related work is discussed in Section 3 and starts with an exhaustive presentation of current and relevant state of the art for VM migration (Section 3.1) and GC (Section 3.2) algorithms. The state of the art discussion finishes with an analysis of resource scalability solutions for Big Data environments in the cloud (Section 3.3).

The global architecture of novaVM is presented in Chapter 4, which is further divided into five main sections, each of which describing the architecture of a sub-component/algorithm (Sections 4.2 to Section 4.6). We conclude this work with implementation details (Chapter 5), evaluation results (Chapter 6), and conclusions and future work ideas (Chapter 7).

Chapter 2

Background

This chapter presents a comprehensive study on several topics that are essential to understand the design decisions supporting this work. It addresses the following issues: i) a study on Big Data environments, ii) the overall architecture behind the JVM runtime system, iii) a theoretical description of memory management, explaining its main concepts and properties, iv) current memory management algorithms while emphasizing why they can not easily scale vertically with regards to memory, v) limitations of current memory management algorithms regarding vertical scalability, and v) a summary of the most relevant concepts here described. Thus, this chapter strives to provide the reader with the necessary background to understand the following chapters of this document.

First, we present a study on Big Data environments (see Section 2.1); this is important to understand the context of the work presented in this document and to further motivate the problems solved by novaVM. Then, we explain the overall architecture behind the JVM runtime system (see Section 2.2). This is essential to understand the basic architecture of the runtime system that is used as a starting point for the contributions presented in this document and also to understand how the proposed profilers will interact with the already existing components in the JVM. Then, we present a theoretical description of memory management, explaining its main concepts and properties (see Section 2.3). The memory management discussion continues on Section 2.4 which shows how the theoretical memory management concepts are applied in real systems by presenting several memory management algorithms that have been proposed and even some that are currently being used in runtime systems such as the JVM. Then (in Section 2.5) we illustrate why current memory management algorithms can not easily scale vertically with regards to memory. The chapter ends with a summary of the most important concepts that are presented.

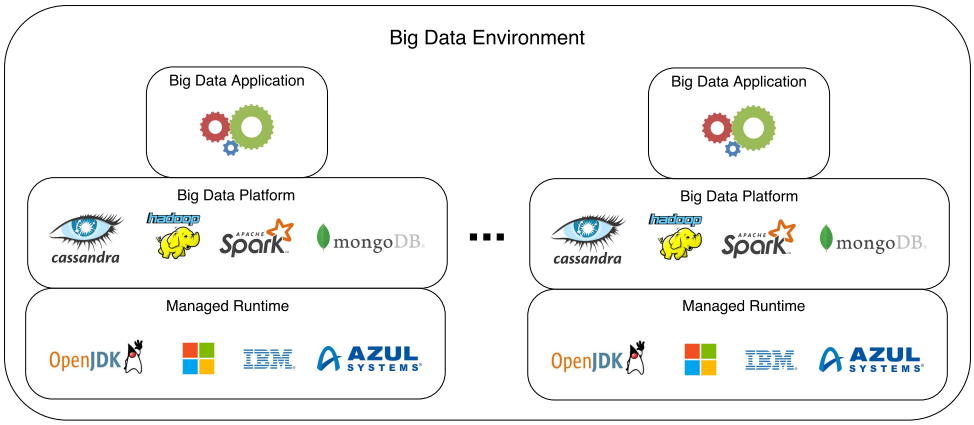

Figure 2.1: Big Data Environment Taxonomy

2.1 Big Data Environments

The term Big Data was used for the first time in an article by NASA researchers Michael Cox and David Ellsworth [33]. The pair claimed that the rise of data was becoming an issue for current computer systems, which they called the ”problem of big data”. In fact, in recent years, the amount of data handled by computing systems is growing. However, not only the amount of data is growing, but also the speed at which it grows is increasing.

Data can be big in many ways. Big Data can be applied in many areas and in each of which it may have slightly different meanings. Within this work, Big Data is used to represent high volumes of data that, because of its dimension, need specialized software tools to handle it (i.e., tools previously developed did not scale, in different performance metrics, to large data sets). The typical motivation for storing and processing such volumes of data is to extract valuable/summarized information from large data in the shortest amount of time.

Throughout this document, we also use the following terms: Big Data environments, Big Data platforms, and Big Data applications (Figure 2.1 illustrates these concepts). The first (Big Data environments) refers to a group of one or more Big Data applications, platforms, and managed runtimes, that are used to complete a specific task. Big Data platforms are frequently organized in a stack, i.e., each platform is given the output of the previous platforms and prepares the input of the next (Figure 2.2 shows an example of such stack: the Hadoop stack). Each Big Data platform represents a processing or storing engine running on top of a managed runtime environment (for example, a JVM). Finally, a Big Data application represents the user code executed by the engine inside the Big Data platform. Some platforms do not take any user code (usually storage platforms); for these platforms, we assume that the application is the platform itself.

The fact that many existing solutions do not scale to Big Data is now widely accepted, as

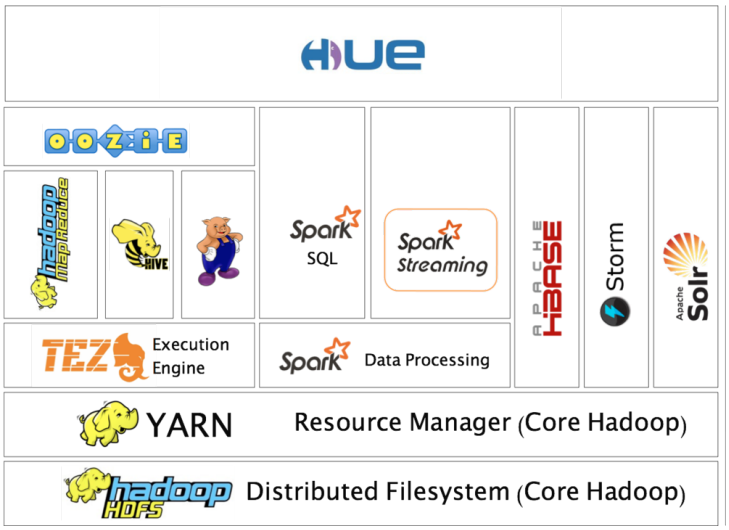

Figure 2.2: Big Data Platform Stack Example: Hadoop Stack

more and more companies invest large amounts of money for creating new Big Data platforms capable of storing and processing their data [25, 87]. Among many real-world Big Data use case scenarios, some of them are the following:

- Trend Analysis. It is known that companies often apply data mining techniques (machine learning, for example) to extract sales patterns associated with some product, advertising, or pricing. This is a clear example where large volumes of data (sales reports in this case) are stored and then processed to extract valuable information. Such information helps company owners adapt their offer to the available market;

- Behavioral Analytics. Similarly to sales report processing, users' information regarding, for example, purchasing or searching habits, can be used to improve their experience by automatically suggesting similar products or results. This is only possible if each user's interactions are recorded and processed to extract his behavior pattern;

- Internet Search Engines. The web crawling process (from which Internet search indexes are built) is another example of a Big Data use case. Companies such as Google, Yahoo!, and Microsoft, process, every day, large amounts of Internet web pages to feed rankings (and other kinds of metrics) to different search engines. In this second scenario, the data is not only analyzed for pattern extraction but also transformed into another representation, one that enables search engines to rank pages according to several desired metrics (keywords, popularity, date of creation, and more);

- Fraud Detection. By extracting user's behavioral information, companies can also detect potential fraudulent behaviors and take preventive measures. Credit card fraud detection

is a real example of this use case. Companies fighting fraud detect unlikely transactions (according to users' behavior and historic) and stop them.

Scalability in Big Data environments is most often measured in terms of throughout and latency scalability. Within this work, we consider that: i) being throughput scalable means that the throughput (number of operations per amount of time) should increase proportionally to the amount of resources added to the system, and ii) being latency scalable means that the latency (duration of a single request) should not increase when the throughput increases. For example, in a fraud detection system, the number of credit card transactions verified per second (throughput) is as important as the duration (latency) of a single credit card transaction verification. Therefore, in the case of fraud detection, the ideal system is both throughput and latency scalable (i.e., the system should increase its throughput as more resources are used but the latency should not be affected by increasing the throughput). However, as we discuss in Section 3.2, optimal throughput and latency can be difficult to achieve at the same time.

The throughput and latency scalability problems are further aggravated if we consider a stack of Big Data platforms (Big Data environment). In this scenario, the throughput of the whole environment is as high as the throughput of the system with lower throughput, and the latency is as low as the system with higher latency. In other words, a single platform can compromise the scalability of the whole environment.

We decompose the challenge of extracting valuable information from very large volumes of information into two sub-problems: i) how to store large amounts of data and provide scalable read and write performance, and ii) how to process large amounts of data in an efficient and scalable way. Both sub-problems are usually handled by different types of platforms: storage and processing. For the remainder of this section, we discuss each type of platform in separate (storage and processing platforms), identifying, for each one, their memory profiles and the resulting challenges, which memory management algorithms are faced with.

2.1.1 Processing Platforms

A Big Data processing platform, in the most simple and generic way, is a system which i) receives input data, ii) processes data, and iii) generates output data. The system can be composed by an arbitrary number of nodes, which can exchange information during the processing stage. Input data can, for example, be retrieved from: i) a storage platform, ii) other processing platforms, or iii) directly from sensors. Output data can be sent to: i) a storage platform, ii) to other processing platforms, or iii) to the final user.

In the remainder of this section, representative real-world Big Data processing platforms

are analyzed. The goal is to understand how these platforms work and, most importantly, how memory is used by these platforms, or, in other words, their memory profile.

MapReduce-based Platforms

MapReduce [34] is a popular programming model nowadays [56, 41]. In a MapReduce application, computation is divided into two main stages: the map stage, and the reduce stage. First, input data is loaded and processed by mappers (nodes assigned with a map task). Mappers produce intermediate data which is then shuffled, i.e., sorted and split among reducers (nodes assigned with a reduce task). In the reduce stage, data is processed into the final output.

Several MapReduce implementations were produced but Apache's Hadoop [123] soon become the de facto standard in both industry and academia [41]. In fact, Hadoop MapReduce is currently used by some of the worlds' largest Information Technology companies, e.g., Facebook [16], Twitter [83], LinkedIn [111], and Yahoo! [107].

The novelty behind recent MapReduce programming model implementations is that most distribution and fault tolerance details are hidden from the programmer. Thus, only two functions need to be defined: i) a map function which converts input data into intermediate data, and ii) a reduce function which aggregates intermediate data. All other steps regarding task distribution, intermediate data shuffling, reading and writing from and to the storage platform, and recovering failed nodes is handled automatically by the platform. Additionally, Hadoop comes with the Hadoop Distributed File System, HDFS, (addressed in Section 2.1.2) which was specially designed to handle large amounts of data. These two systems (Hadoop MapReduce and HDFS), while working together, can be used to create a Big Data environment with both processing and storage capabilities.

Another important factor about MapReduce and HDFS is that both platforms are basic building blocks for more complex Big Data environments (these platforms represent stack layers on top of MapReduce and HDFS) such as Hive [115], Pig [97], and Spark [128]:

- Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. It provides an SQL-like language called HiveQL which automatically converts queries into MapReduce jobs that can be executed in Hadoop. All data is read from and written to HDFS;

- Pig is a high-level platform for creating MapReduce programs in Hadoop. The language for this platform is called Pig Latin; it abstracts the programming model from the Java MapReduce idiom into a notation that makes MapReduce high-level programming, similar

to that of SQL for RDBMS systems. Similarly to Hive, input and output data comes and goes to HDFS and Hadoop MapReduce is used to perform the MapReduce tasks;

- Spark is a MapReduce engine (among other capabilities) that enables efficient in-memory data processing. Spark aims at improving the performance of applications that reuse data between MapReduce cycles. In Hadoop, between each cycle, all data must be flushed to disk (HDFS) and retrieved in the next MapReduce cycle. To cope with this performance drawback, Spark provides the Resilient Distributed Dataset (RDD) which maintains a collection of objects that can be used in subsequent MapReduce iterations. Spark is very popular, for example, in iterative machine learning algorithms.

Directed Graph Computing Platforms

Another relevant type of processing platforms to consider is the one based on directed graphs. This is a more general model than MapReduce model as it allows arbitrary flows of data among computations. The model uses directed graphs to express data processing tasks (vertexes), and data dependencies (edges). The developer is left with the job of building the computation graph and providing a function to execute on edges.

Several Big Data platforms have been proposed using graphs to express computations and data flows. The goal for the rest of this section is to analyze some well-known platforms, namely Dryad [127], Naiad [93], Pregel [89], and MillWhell [2], in order to understand their memory profiles. Although these platforms might not run on top of the same runtime environment (some platforms might run on top of a JVM, others might run on top of the .NET Common Language Runtime [17]), all runtime systems must deal with automatic memory management, which is directly affected by the memory profile of each platform.

Dryad is a general-purpose distributed execution engine for data parallel applications. Dryad allows the definition of applications that organize computation in edges and communication as data channels. The platform provides automatic application scheduling, handles faults, and automatically moves data along edges into vertexes. Dryad application developers can specify an arbitrary directed acyclic graph to describe application's communication patterns, and express the computation that takes place at vertexes using subroutines.

Naiad is a distributed system for executing data-parallel, cyclic dataflow programs. Therefore, as opposed to Dryad, it allows the definition of directed graphs with cycles. The main goal of Naiad is to provide a platform which processes a continuous incoming flow of data and to allow low-latency, on-the-fly queries over the processed data. This is specially important for streaming data analysis, iterative machine learning, and interactive graph mining. Similarly to

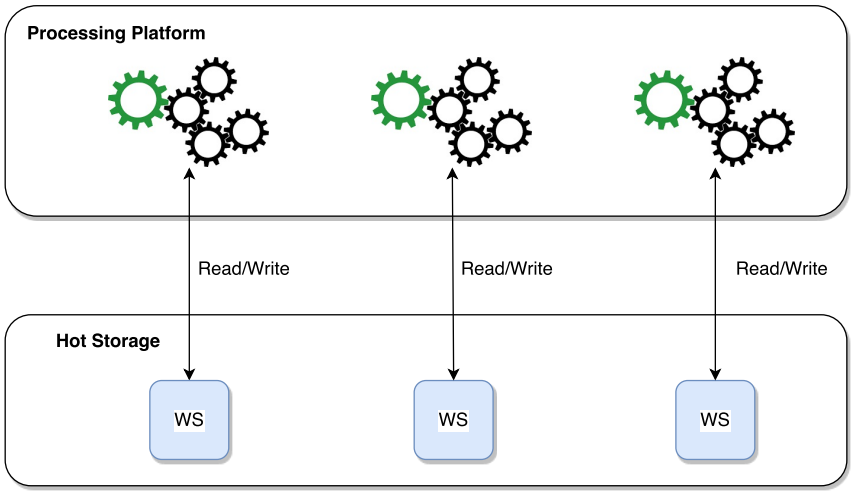

Figure 2.3: Typical Processing Platform

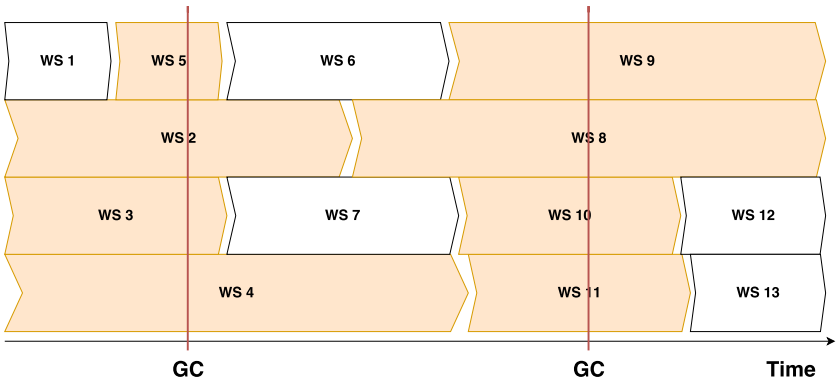

Figure 2.4: Processing Platform Working Sets

Dryad, communication between vertexes can be implemented automatically (this is typically left for the developer to decide) in multiple ways.

MillWhell is a similar approach, compared to the previous solutions, for low-latency data streaming platform. It also provides developers with the abstraction of directed computing graphs which can be built using arbitrary and dynamic topologies. Data is delivered continuously along edges in the graph. MillWhell, similarly to the previous approaches, provides fault tolerance at the framework level (i.e., the programmer does not have to deal with faults, the framework automatically handles them).

Processing Platforms Memory Profile

All processing platforms discussed so far can be reduced to a model where an arbitrary set of nodes perform some computation (task) and data flows (in and out) among computing nodes. Therefore, since most processing platforms can be reduced to a common representation, most of the problems of a specific platform will apply to the other processing platforms.

The memory profile for processing platforms is very characteristic. Each task usually has a Working Set (WS) which is loaded into memory (the WS can vary in size according to each platform and application), and is used to read and write over working data (see Figure 2.3). Each WS is specific to a single task and therefore, is considered garbage after the task is finished. If multiple tasks run in parallel, multiple WSs will be present in memory at the same time. Finally, each task can have different execution times, resulting in different WSs being present in memory for different amounts of time (Figure 2.4 illustrates this situation). If a GC is triggered, all WSs currently being used (represented as pink boxes in Figure 2.4) will be handled by the collector, and thus, all data within is going to be moved to other memory location, producing a severe throughput degradation and high application latencies. White boxes represent WSs that are not being used when the GC is triggered and therefore are ignored by the collector.

The practical effect of this problem is present in many platforms. These platforms suffer from high GC interference, hindering application throughput and latency. For example, in a platform with multiple tasks, each with dependencies from other tasks, GC can turn nodes into computation stragglers [95]. Other consequence is the increased latency, result of long GC pauses (as demonstrated through performance experiments in Sections 6.3, 6.4, and 6.5). In Section 2.4, we further discuss this issue: why processing platforms' memory profile stresses memory management, leading to throughput and latency scalability problems.

Programmers try to reduce the GC interference by using a number of techniques such as: i) delaying collections as much as possible, or ii) using pools of objects than can be reused multiple times (to reduce object allocations), or iii) serializing multiple objects into very large arrays of bytes. These solutions, as we discuss in Section 2.4, have very limited success.

2.1.2 Storage Platforms

A storage platform, in the most generic way, is a system that provides read and write operations to some managed storage. The platform can orchestrate a number of nodes to store data. Each node provides volatile but fast storage, and persistent but slow storage. Read and write operations may obey a variety of consistency models [80] (this topic, however, is not in the scope of this work).

In the remainder of this section, some of the most important types of storage platforms are analyzed. The goal is to understand how these platforms work and, most importantly, their memory profile.

Distributed File Systems

A Distributed File System (DFS) is a storage system in which files are accessed using interfaces and semantics similar to local file systems. Therefore, DFSs normally provide a hierarchical file organization which can be accessed using basic file system primitives such as open, close, read, and write. It is often the case that DFSs can be mounted on the local file system.

The Hadoop Distributed File System (HDFS) is a popular example of a Big Data DFS. Inspired by the Google File System [49], HDFS aims at providing an efficient approach to access large-scale volumes of data. It uses a centralized entity that stores metadata and many data nodes to store files. HDFS, which integrates with Hadoop MapReduce, also employs several important performance optimizations that do not fall within the scope of this document.

Graph Databases

A graph database is a storage system that provides a graph management interface for accessing graphs stored within it [104]. With the growth of data, many companies soon started to represent their application domains using graphs, which, for some applications such as social networks, gives a much more intuitive representation than other data models. Additionally, these systems provide efficient graph computing/search engines which enable applications to perform queries or even modify the graph in a very efficient and scalable way. Several graph databases have been developed but, without loss of generality, only two representative examples are addressed in this section.

Titan[1] is a distributed graph database featuring scalable graph storing and querying over multi-node clusters. Titan is a very versatile solution as it can use several storage back-ends, for example Cassandra (see Section 2.1.2), and exports several high level APIs. Titan is often used with Gremlin[2], a graph traversal language.

Another example of a graph database is Neo4J [118]. Opposed to Titan, Neo4J is a centralized graph databases that offers applications primitives to build and manage graphs.

Key-Value and Table Stores

The last two types of Big Data storage platforms to consider are key-value stores and table stores. For the sake of simplicity, and without loss of generality, within this section, it is assumed that the only difference between both types is the way data is presented to the application (i.e., the interface): as a distributed key-value store, (similar to a Distributed Hash Table), or

[1] Titan's web page can be accessed at http://thinkaurelius.github.io/titan/
[2] Gremlin's web page can be accessed at https://github.com/tinkerpop/gremlin/wiki

as a table store (in which information is formatted in rows and columns). Additionally, only two representative platforms are discussed: HBase [48] (based Google's BigTable [27]), and Cassandra [79]. There are many other platforms (such as Dynamo [35] , OracleDB [3], MongoDB [28], etc.) but those are not addressed since the principles behind all these solutions are similar.

HBase is a distributed table-oriented database. It is inspired by Google's BigTable and runs on top of Hadoop MapReduce and HDFS. HBase provides strictly consistent data access and automatic sharding of data. HBase uses tables to store objects in rows and columns. To be more precise, applications store data into tables which consist of rows and column families containing columns. Each row may have different sets of columns, and each column is indexed with a user-provided key and is grouped into column families. Also, all table cells are versioned and their content is stored as byte arrays.

Cassandra is a distributed key-value store. It is designed to handle large amounts of data spread out across nodes while providing a highly available service with no single point of failure (as opposed to HBase, Cassandra has no centralized master entity). The major difference between Cassandra and HBase lies on the data model provided by both solutions. Cassandra provides a key-value store where columns can be added to specific keys. In Cassandra, one cannot nest column families but can specify consistency requirements per query (which is not possible in HBase). Moreover, Cassandra is write-oriented (i.e., the platform is optimized for write intensive workloads) whereas HBase is designed for read intensive workloads.

Storage Platforms Memory Profile

In general, storage platforms take advantage of fast/hot storage to keep caches of recently read or written objects while all remaining objects are stored in slow/cold storage (see Figure 2.5).

Similarly to processing platforms, storage platforms have a very specific memory profile. These platforms tend to cache as many objects in hot storage (usually DRAM) as possible in order to provide fast data access and to consolidate writes. For example, in Cassandra, the result of write operations is cached in large tables in memory in the hope that future read or write operations will use the same result (thus avoiding a slower access to disk). Multiple caches can coexist in memory at the same time and may have different eviction policies (usually limited by the available memory). According to our experience, caching data in memory to avoid slow disk accesses and to consolidate writes is a frequent technique across many storage platforms.

By aggressively caching data, storage platforms keep many live (reachable) objects in memory, leading to severe GC effort to keep all objects in memory (this problem is further discussed

[3] OracleDB's web page can be found at https://www.oracle.com/database/index.html

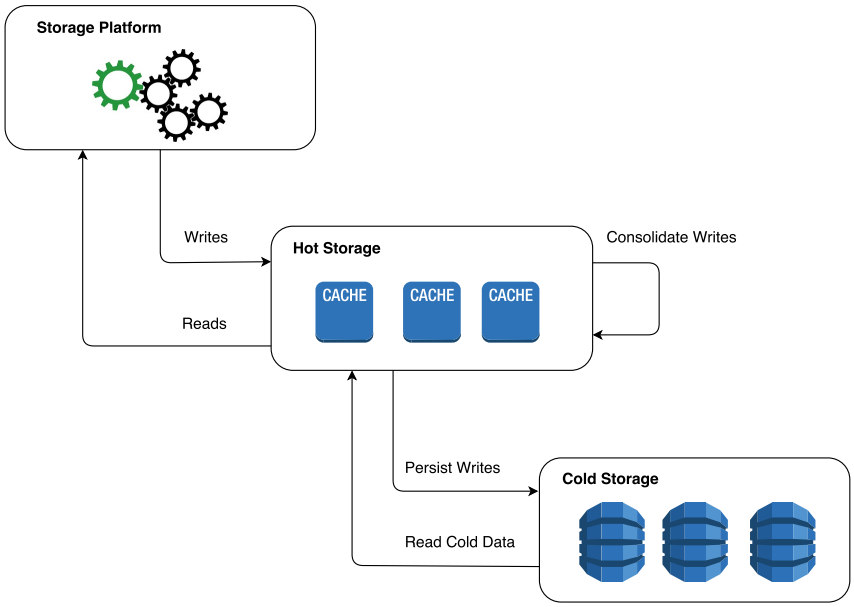

Figure 2.5: Typical Storage Platform

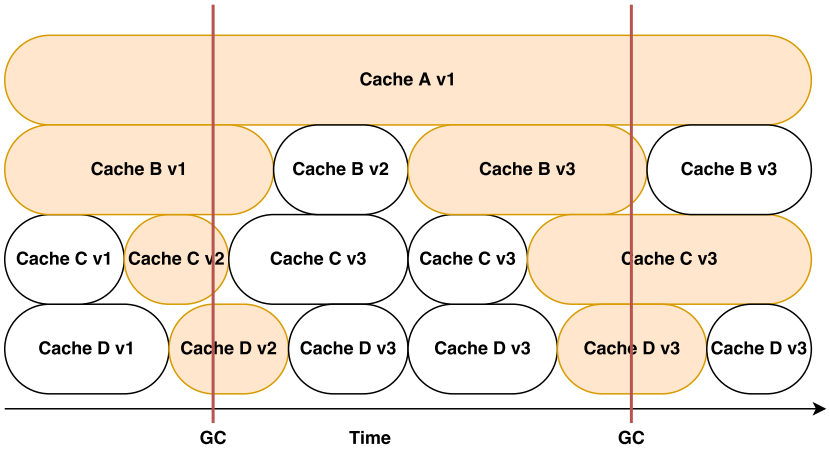

Figure 2.6: Storage Platform Caches

in Section 3.2). This leads to the same problem discussed in Section 2.1.1, i.e., during a collection, all objects belonging to all active caches will be handled by the collector (they are moved to other memory location). Figure 2.6 illustrates this problem; active caches upon collection (represented in pink) will be moved to other memory location (caches represented in white are not being used anymore and therefore are not considered by the collector). In this scenario, GC will lead to long applications pauses, directly increasing the platform latency (for example, read or write operation latency in Cassandra) and reducing throughput (as demonstrated through performance experiments in Sections 6.3, 6.4, and 6.5). In Section 3.2, we further discuss this issue: why storage platforms' memory profile stresses memory management, leading to latency and throughput scalability problems.

Naive solutions such as such as i) severely limiting the size of the heap, and/or ii) reducing

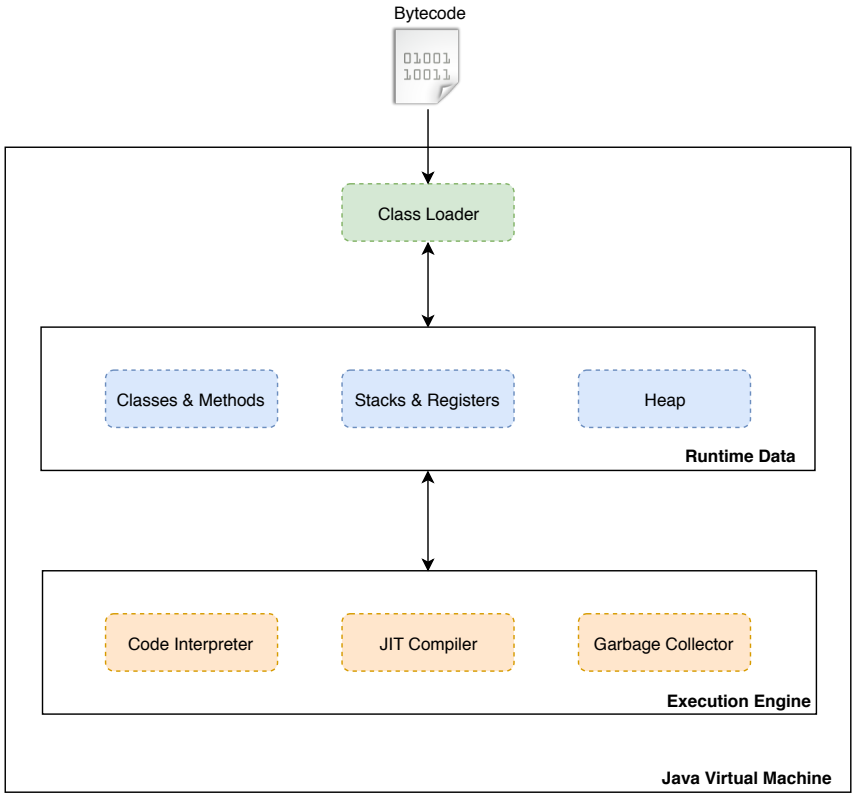

Figure 2.7: OpenJDK HotSpot JVM Architecture

the number of requests to handle per second, will not only reduce the throughput but will not solve the problem (i.e., it will only soften its effects).

2.2 JVM Architecture

As discussed in the previous section, Big Data environments comprehend a number of Big Data platforms and applications. These platforms and applications typically run on top of a runtime system, a middleware layer that abstracts the underlying system where Big Data platforms and/or applications run.

We dedicate this section to study the architecture and techniques present in most runtime systems, with special focus on the OpenJDK HotSpot JVM, the baseline JVM on top of which novaVM is designed and implemented.

Having a good background in runtime systems is crucial to follow the work presented in this document as most of the developed algorithms are designed and implemented to fit inside a runtime system and may even require coordination with other components already present in the runtime system.

Figure 2.7 presents the high level architecture of a runtime system. Note that Figure 2.7 represents the architecture of the OpenJDK HotSpot JVM but many other runtimes share most

of the components and techniques, if not all. Additionally, it presents a simplified architectural overview. The elements that are not present are not relevant for the context of this work and therefore, presenting them would overcomplicate the architectural description with no benefit.

Starting from the top of Figure 2.7, many high-level languages compile to bytecode, a program representation that uses an instruction set designed to provide efficient execution using a software interpreter (software that reads the bytecode and executes the operations described in it). Besides being designed for efficient interpretation, the bytecode representation also provides hardware independence, since the instruction set used to create the bytecode is hardware agnostic. In order to compile a program from source code for example, written in Java, to bytecode, languages provide compilers such as the Java Compiler (`javac`).

After compiling the program into bytecode, the runtime system (the JVM, in this case) can execute the bytecode. To do so, the runtime uses a Class Loader, a component that loads bytecode and prepares the necessary runtime data structures to execute the bytecode.

These data structures that support the execution of the application are included in one of the main component groups in the runtime architecture: Runtime Data. The Runtime Data is comprised by a number of data structures that support the execution of the program. In this work, we bring special focus to some of these data structures:

- Classes & Methods - these data structures maintain, inside the runtime system, the representation of classes of objects and methods defined in the program whose bytecode is given as input;
- Stacks & Registers - these data structures maintain the execution state of each program thread. Note that this information is very important when the runtime system needs to interrupt program threads to perform operations such as collecting garbage. In such scenarios, these data structures hold the threads' state that will allow the runtime system to resume the program's execution;
- Heap - this data structure is where memory that is being used by the program is kept. This data structure is further analyzed in the next section.

All the previously presented data structures are necessary to run and keep track of the program's execution, which is controlled by the components in the Execution Engine. In the Execution Engine, we bring to focus the following components:

- Code Interpreter - the Code Interpreter is the software component that interprets bytecode and, depending on the instruction and arguments, it will execute the correspondent operation in the hardware-specific instruction set;

- JIT Compiler - the JIT compiler is a component that converts bytecode into native, hardware-specific code. A JIT compiler is used to compile highly executed code into native code, which is much faster to execute compared to interpreted code. Besides compiling bytecode to native code, a JIT compiler will also perform optimizations such as method inlining, branch prediction, among others;
- Garbage Collector - the Garbage Collector is the component whose main task is to manage memory. This component is discussed in greater depth in the next section.

To summarize, runtime systems allow Big Data platforms and application developers to express their programs using high-level languages which abstracts/simplifies memory management and provides greater code portability. In order to do so, runtime systems, and the OpenJDK HotSpot JVM in particular, rely on a set of Runtime Data structures and on a complex Execution Engine containing a Code Interpreter, a JIT Compiler, and a Garbage Collector.

2.3 Memory Management Background

As introduced in the previous sections, memory management (both automatic and manual) deals with two important problems: i) provide memory when requested (memory allocation), and ii) free unused memory for future reuse (memory deallocation). This is a classical problem in every Operating System (OS) [112]. Nevertheless, memory management is also a fundamental problem for the JVM since it has to automatically manage memory, which is previously allocated by some underlying OS, and is necessary for the end-user application to run (it acts as an intermediary management system between the underlying OS and the application).

2.3.1 Background Concepts

Before delving further on how memory is managed by the JVM, it is important to provide some background concepts and explain how memory is structured in a Java application. The first important concept to introduce is the heap. From the JVM point of view, a heap is a contiguous array (or set of arrays) of memory positions which may be occupied or free. These memory positions are used to store objects. An object is a contiguous set of memory positions allocated for the end-user application to use. An object is divided into fields (or slots) containing a reference or some other scalar non-reference type (an integer, for example). A reference is either a pointer to a heap object or the distinguished value, *null*.

In applications, many objects populate the heap. Therefore, the heap is often characterized as a directed graph where nodes are application objects and edges are references from other

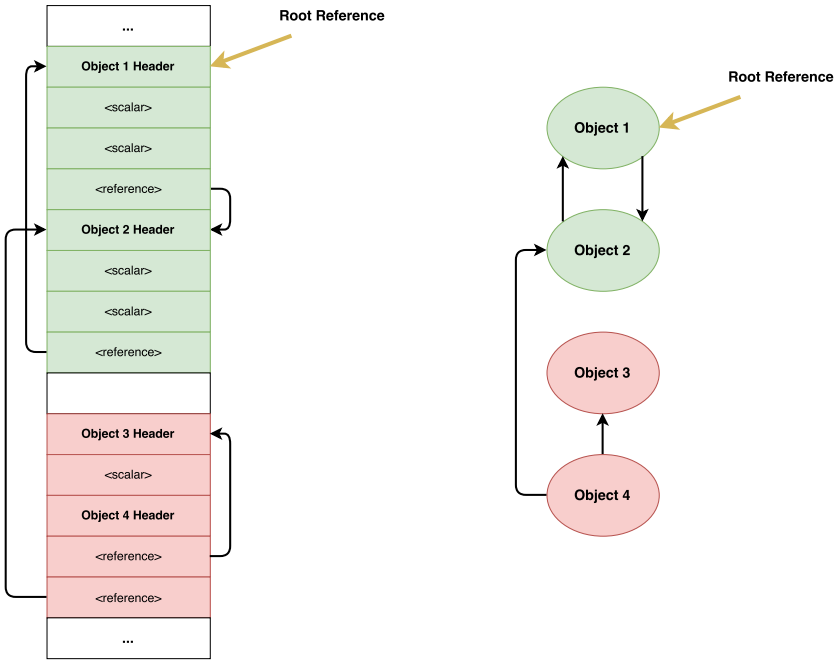

Figure 2.8: Java Memory Heap (left) and the corresponding Java Object Graph (right)

objects (or from a root). A root is a reference held by the JVM that points to an object inside the object graph. There are several root references and the objects pointed by these references are named root objects. Examples of root references include global variables, and variables held in CPU registers.

Objects in the object graph can be identified as reachable (or live) or unreachable (or dead). An object is said to be live if there is a path of objects (such that a reference from one to the other exists) starting from any root object that reaches the object. On the other hand, if there is no path from any root object to an object, the object is considered dead and its memory should be collected for future reuse.

Figure 2.8 presents the concepts just introduced. The Java heap is presented as a continuous array of memory positions with objects. Each object contains several fields with scalar or reference types. The corresponding object graph is presented on the right. Reachable (live) objects are represented in green while unreachable (dead) objects are represented in red.

Following the terminology introduced by Dijkstra [39], a garbage-collected program is composed by two elements: i) a mutator, and ii) a collector. The mutator represents the user application which mutates the heap by allocating objects and mutating these objects (changing references and fields). On the other hand, the collector represents the garbage collector code which manages memory.

2.3.2 GC Properties

There are many GC algorithms, with different implementations, many of which using different approaches to collect dead objects. To better understand the desirable properties and trade-offs of each GC algorithm, it is important to point out the most critical properties/factors that can be considered when comparing the performance of different GC algorithms:

- Safety. A safe collector never reclaims the storage of live objects;
- Throughput. Throughput is commonly associated to the number of operations that can be executed in a particular execution or interval of time. The goal for every GC is not to decrease the application throughput but to increase it if possible (compared to using manual memory management);
- Completeness. A complete collector is one such that all garbage is eventually reclaimed. This is a special concern for reference counting approaches (described in Section 2.4.2), which are typically not complete;
- Promptness. High promptness means that the collector takes little time to reclaim memory used by an object after it becomes unreachable. On the other hand, low promptness characterizes GCs which take a long time to reclaim memory occupied by objects that are no longer reachable;
- Pause time. How much time the application must stop to let the GC execute. The time that the application is stopped is called pause time. For the duration of the pause time, no application code can be running;
- Space overhead. The amount of space required to perform GC. For example, copying collectors usually need more memory (to perform a clean copy) than compacting collectors (which only move live objects to the beginning of the heap);
- Scalability. A GC is considered scalable if an increase in the number of objects in memory does not compromise performance decrease in any of the previous metrics.

2.4 Classic Garbage Collection Algorithms

Having described the basic concepts regarding GC, this section is dedicated to studying the local GC problem, i.e., memory management performed on a single process' address space. The study of distributed memory management, distributed garbage collection, is left out of the

scope of this work since, according to our experience, no significant Big Data platform is using such type of algorithms.

We start our study by presenting the most commonly used algorithms for allocating memory. Then we present the three main families of collectors (reference counting, tracing, and partitioned/hybrid), followed by a discussion of typical GC design choices.

2.4.1 Memory Allocation

The two main aspects of memory management, memory allocation and memory reclamation, are tightly linked in a sense that the way memory is reclaimed places constraints on how it is allocated and vice-versa. While a memory allocator needs a collector to identify free memory positions, the collector may also need the allocator to provide free memory to enable some GC operation (see Section 2.4.4 for more details).

Sequential allocation is the first and most simple allocation algorithm. It uses a large free chunk of memory from the heap. Only two pointers need to be maintained between allocations: a free pointer (which limits the last allocated fraction of the heap), and a limit pointer (which points to the last usable memory position). When an allocation takes place, the free pointer is incremented by the number of requested blocks. If there are not enough blocks between the free pointer and the limit pointer, an error is reported and the allocation fails. This algorithm is also called bump pointer allocation because of the way it ”bumps” the free pointer.

Despite its simplicity, the algorithm is efficient and provides good locality [11]. However, as time goes by, some objects become unreachable while others are still reachable. This results in many small allocated blocks interleaved with many unallocated blocks, i.e., high fragmentation.

The alternative to sequential allocation is free-list allocation. In a basic free-list allocation algorithm, a single data structure (a list) holds the size and location of all the free memory blocks. When some memory is requested, the allocator goes through the list, searching for a block of memory that fits the requested size, and respecting an allocation policy. There are many allocation policies and therefore, only the most used ones are presented described:

- first-fit, the simplest approach. The allocator stops searching the list when it finds a block with at least the required number of memory blocks. The allocator might split the free chunk in two if the chunk is larger than required;

- next-fit, a variant of the first-fit algorithm. It starts searching for a block of suitable size where the last search ended. If the allocator reaches the end of the list, it restarts the search from the beginning of the list.

- best-fit finds the free block whose size if the closest to the request. The idea is to minimize memory waste and avoid splitting large memory blocks unnecessarily. This is the policy behind the well know Buddy [76] allocation algorithm.

Even with free-list allocation, the user might notice that the time it takes to allocate memory is linear with the size of the memory (heap). If the size of the memory grows significantly, the time needed to allocate some blocks will become prohibitive. To cope with this problem, there are some optimizations. The first optimization consists on using balanced binary trees to improve worst-case behavior from linear to logarithmic in the size of the heap. Hence, instead of going through all elements of a list, the allocator can traverse the tree searching for the block with the requested size. This technique is also known as fast fit allocation [110].

The second optimization comes from the fact that much of the time consumed by a free-list allocator is still spent searching for free blocks of appropriate size. Therefore, using multiple free-lists, whose members are grouped by size, can speed allocation. By using enough lists with appropriate block ranges it is possible to achieve allocation in almost constant time.

So far, the described techniques and algorithms manage the whole heap, i.e., if some memory needs to be allocated, the allocator must preserve the integrity of all allocation data structures by using atomic operations or locks. In a highly threaded environment, this is a serious bottleneck. The common solution to cope with this problem is to give each thread its own allocation area, the Thread Local Allocation Buffer (TLAB). This way, each thread can allocate memory from its TLAB independently, i.e., with no synchronization. Threads may always request new TLABs from the heap if the current TLAB runs out of memory. Only interactions with the global memory pool (heap) are synchronized (e.g., if some object does not fit the TLAB, it must be allocated directly on the heap). Dimpsey [40] measured substantial performance improvements in a multi-threaded Java system using a TLAB for each thread. It was also possible to conclude that TLABs tend to absorb almost all allocation of small objects. Most accesses to the heap turned out to be requests for new TLABs. TLABs can be used for sequential allocation or free-list allocation.

Having described how current GC implementations handle memory allocation, the next sections are focused on how memory is reclaimed.

2.4.2 Reference Counting Algorithms

As the name suggests, reference counting algorithms (first introduced by Collins [32]) literally count references to objects. Such algorithms are based on the following invariant: an object is considered alive if and only if the number of references to the object is greater than zero

Algorithm 1 Reference Counting

```
1:  procedure ALLOCATE(objType)
2:      object ← allocateObject(objType)
3:      resetCounter(object)
4:      return object

5:  procedure ATOMIC MUTATE(parent, slot, newChild)
6:      AddReference(newChild)
7:      DelReference(parent.slot)
8:      parent.slot = newChild

9:  procedure ADDREFERENCE(object)
10:     incrementCounter(object)

11: procedure DELREFERENCE(object)
12:     decrementCounter(object)
13:     if getCounter(object) == 0 then
14:         for child in childRefs(object) do
15:             DelReference(child)
16:         free(object)
```

(note that these algorithms erroneously consider objects included in cycles of garbage as live objects). Therefore, to be able to know if an object is alive or not, reference counting algorithms keep a reference counter for each object. Reference counting is considered direct GC (as opposed to indirect GC which is discussed next) since it identifies garbage, i.e., objects with no incoming references.

Algorithm 1 presents a reference counting algorithm. It is important to note that the Mutate operation must be atomic, i.e., there should be no other Mutate operations executing at the same time. Otherwise, counters could be erroneously updated.

Contrary to reference tracing algorithms (see Section 2.4.3), reference counting algorithms provide some interesting properties: i) the GC overhead is distributed throughout the computation, i.e., it does not depend on the size of the heap, but, instead, on the amount of work done by the mutator; ii) garbage can be collected almost instantaneously (as the collector knows instantly when the number of incoming references reaches zero); and iii) it preserves cache locality (by not traversing the object graph and therefore destroying the application working set cache locality).

These advantages come with two drawbacks: i) high overhead of maintaining track of a counter for each object (which incur into synchronized operations whenever it needs to be updated); and ii) reference counting is not complete, i.e., not all garbage is collected (particularly, cyclic garbage).

To cope with the first drawback, Blackburn et al. [12] propose a useful taxonomy of solutions:

- deferred reference counting, delay the identification of garbage to specific periodic checkpoints. This way, some synchronization steps are avoided;
- coalescing, a technique based on the hint that many reference count adjustments are temporary and therefore, can be ignored (for example, GC operations on local variables). With coalescing, only the first and the last state of an object field should be considered. Reference counting increments or decrements should only be considered at specific checkpoints, thus safely discarding many other intermediary states;
- buffered reference counting, in which all reference count increments and decrements are buffered for later processing.

All these three approaches try to reduce some of the synchronization overhead inherent to updating global reference counters. To deal with the second drawback (completeness), the most widely used solution is to perform trial deletion. Trial deletion is a technique that requires a backtracking algorithm to visit objects that are suspected to contain cyclic garbage. The main idea behind the algorithm (described by Paz et al. [99]) is to check if cyclic garbage is uncovered when some pointer is deleted. If the reference count of the object whose pointer is deleted reaches zero, it exposes the existence of cyclic garbage.

2.4.3 Reference Tracing Algorithms

Reference tracing algorithms rely on traversing the object graph and marking reachable objects. Reference tracing is quite straightforward; objects that are marked during reference tracing are considered alive. All memory positions that are not marked, are considered to be garbage and will be freed. Hence reference tracing is considered indirect GC, i.e., it does not detect garbage but live objects instead. Typical implementations of reference tracing collectors are also known as mark-and-sweep collectors [91].

Algorithm 2 presents a mark-and-sweep collector. Despite its simplicity, this mark-and-sweep algorithm has some problems regarding the need to stop the mutator from changing the object graph during marking (this is discussed in more detail on Section 2.4.4). To cope with this problem, a second mark-and-sweep implementation, which uses the tri-colour abstraction [39], is used. This approach, also called tri-color marking, provides a state for each object in the object graph. Hence, each object can be in one of the following states:

- white, object not reached, the initial state for all objects;

Algorithm 2 Mark and Sweep

```
 1: procedure MARKROOTS
 2:     for object in roots do
 3:         if notMarked(object) then
 4:             setMarked(object)
 5:             push(objStack)
 6:             Mark

 7: procedure MARK
 8:     while object in objStack do
 9:         object = pop(objStack)
10:         for child in childRefs(object) do
11:             if notMarked(object) then
12:                 setMarked(object)
13:                 push(objStack, child)

14: procedure SWEEP(heapStart,heapEnd)
15:     curr = nextObject(heapStart)
16:     while curr < heapEnd do
17:         if isMarked(curr) then
18:             unsetMarked(curr)
19:         else
20:             free(curr)
21:         curr = nextObject(curr)
```

- black, object that has no outgoing references to white objects. Objects in this state are not candidates for collection;

- gray, object that still has references to white objects. Gray objects are not considered for collection (eventually, they will turn black).

Tri-color marking starts by placing all root objects in the gray set (set of gray objects) and all remaining objects in the white set (set of white objects). The algorithm then proceeds as follows: while there are objects in the gray set, pick one object (from the gray set), move it to the black set (turning it into a black object), and place all objects that it references in the gray set (turning objects into gray objects). In the end, objects in the black set are considered alive. All other objects (white objects) can be garbage-collected. Using the aforementioned steps, the algorithm keeps the following invariant: no black object points directly to a white object.

By using a state for each object, the collector can remember which objects were already verified and therefore, it can run incrementally or even concurrently with the mutator. However, care must be taken to track situations where the mutator writes a reference to a white object into a black object (this would break the algorithm invariant).

A final remark about this algorithm is that, although sweeping needs to search the whole

heap (for collecting white objects), this task can be delayed and performed by the allocator [62] (this technique is called Lazy-sweeping).

2.4.4 Design Choices

Both approaches for GC, tracing and reference counting, can be designed and optimized for different situations. For example, in a multi-core architecture, one would want to take advantage of multiple cores to split the GC task among several cores (to achieve higher performance). Another interesting and challenging scenario is to run the GC concurrently with the application (mutator). In a multi-core architecture, mutator threads can run concurrently with collector threads, therefore increasing the application responsiveness and decreasing the pause times. Yet another possible optimization is to periodically clean (by copying or compacting) areas of memory with low live data or high fragmentation.

Hence, each of the previously presented approaches to GC (tracing or counting) can be customized according to several design choices:

- Serial versus Parallel — The collection task can be executed by one or several threads. For example, in reference tracing, traversing an object graph can be done in serial (single thread) or in parallel mode (multiple threads). It is clear that a parallel implementation of either reference tracing or reference counting can harness multiple execution flows on available CPU cores but it also requires a more careful implementation due to complex concurrency issues;

- Concurrent versus Incremental versus Stop-the-World — Stop-the-World GC means that most of the GC work is done when no mutator task is running. This means that all application threads are stopped periodically to enable GC to run. To minimize the time the application is stopped (pause time), one could implement an incremental GC, in which the collection is done in steps, e.g., per memory page, per sub-heap, per sub-graph. If the goal is to mitigate application pauses, it is possible to implement a concurrent GC, where both mutator and collector run at the same time.

  It is important to notice some trade-offs regarding GC implementations. Stop-the-World implementations are the simplest because there is no need to synchronize mutator and collector threads. Yet, it is the best option for throughput oriented applications because it does the collection in only one step, and lets the application run at full speed the rest of the time. The same is not true for incremental or concurrent GCs. These are targeted to applications with low latency requirements. As the collection is done in steps, overall it

might require more time to collect all garbage. The necessary synchronization between mutator and collector threads is also a source of overhead compared to Stop-the-World implementations. The use of read barriers [7] and/or write barriers [94] are common approaches to synchronize collector operations and mutator accesses to objects being collected. In both approaches, some mutator's reads and/or writes are checked for conflicts before the operation takes effect.

- Compaction versus Copying versus Non-Moving — The last design decision is about whether or not to move live objects in order to reduce fragmentation. Fragmentation occurs when objects die and free space appears between live objects. The problem is that, with time, most free memory is split into very small fragments. This leads to three serious problems: i) locality is reduced, i.e., objects used by an application are scattered through all the heap; ii) objects which cannot fit inside memory fragments cannot be allocated; iii) the total amount of memory used by an application is high (since fragments between live objects force the application to use more memory to keep creating objects).

  To solve the fragmentation problem, two typical solutions can be employed: i) compaction, and ii) copying. Both techniques require live objects to be moved and grouped to reduce fragmentation. Compaction is frequently used to move all live objects to the start of some memory segment (for example, a memory page); copying, on the other hand moves live objects from one memory segment to another. Although requiring more memory, copying allows an application to group objects from multiple memory pages (with few live objects) into a single page. Pages from where objects were copied can be freed. The same does not occur with compaction, where multiple pages with few live objects can still coexist.

  The decision of when to apply compaction or copying is also an interesting research problem (that falls outside the scope of this work). Typical solutions involve measuring the percentage of: i) fragmentation, ii) live objects, and iii) memory usage for each memory segment. Only if there are few live objects or high fragmentation, the cost of copying or compacting will compensate the overhead of moving live objects [108].

2.4.5 Partitioned/Hybrid Algorithms

So far, only monolithic approaches to GC have been described, i.e., the whole heap is collected using one GC algorithm only. However, nothing prevents heap partitioning into multiple partitions/sub-heaps and apply different GC approaches on each sub-heap. The motivation behind these hybrid algorithms resides in the fact that, different objects might have different

properties that could be explored using different GC approaches.

The idea of heap partitioning was first explored by Byshop [10]. With time, several partitioning models have been proposed:

- partitioning by mobility, where objects are distinguished based on their mobility, i.e., objects that can be moved and objects that can not be moved or are very costly to move;
- partitioning by size, where objects of certain dimensions are placed in a separate object space, to prevent or minimize fragmentation;
- partitioning for space, where objects are placed in different memory spaces so that the overhead applying GC techniques such as copying can be reduced. To this end, each memory space can be processed separately;
- partitioning by kind, where objects can be segregated by some property, such as type. This can offer some benefits as properties can be assessed using the object's memory address (thus avoiding loading the object's header from memory);
- partitioning for yield, the most well-known and widely used partitioning technique, where objects are segregated to exploit their life cycles (i.e., group objects by their estimated life time). Studies have confirmed that Java object's life time follows a bimodal distribution [66, 68] and that most objects die young [116];
- partitioning by thread, where objects are allocated in thread-local heaps, similar to a TLAB [65, 47]. Such object placement leads to high concurrency improvements since only one mutator thread must be stopped at each time to collect garbage.

For the remainder of this section, a deeper look is taken at the most used type of heap partitioning, generational GC [82, 42, 5, 103, 124, 43, 6], where partitioning considers the age of objects. As already discussed before, Java objects' life time tends to be split between long lived objects and short lived objects. Using this property, it is possible to split objects according to their life cycle and use different sub-heaps (or generations) for long and short lived objects. The age of an object corresponds to the number of collections the object has survived.

Considering that short lived objects turn into garbage very soon, the young generation (sub-heap where short-lived objects are placed) will most likely be with very few live objects very quickly. On the other hand, the old generation will take much longer to accumulate garbage. Using this knowledge, generational GCs are able to reduce an application's pause time by collecting more often the young generation (which is usually small) and collecting less often the old generation (which is usually large).

Generational collection can improve throughput by avoiding processing long-lived objects too often. However, there are costs to pay. First, any old generation garbage will take longer to be reclaimed compared to garbage in the young generation. Second, cycles with objects in multiple generations might not be reclaimed directly (as each GC cannot determine if references going to other generations are part of a cycle). Third, generational GCs impose an overhead on mutators in order to track references that span generations, an overhead hoped to be small compared to the benefits. For example, in a scenario with only two generations (young and old), these references are typically coming from old to young generation and therefore are part of the young generation root set (also called remember set), necessary to allow the young generation to be collected independently from the old generation. These references can be maintained by using a write barrier [117, 92, 5] or indirect pointers [82].

To deal with the possible high pause time coming from old generation collections, which might be high, Hudson et al. [61] propose a new approach, the train algorithm. In this algorithm the old generation is divided into cars (memory segments) of fixed memory size. GC collects at most one car each time it runs. Additionally, objects are moved (from one car to another) in order to cluster related objects. When some car is empty, it can be recycled. This way, using the train algorithm, application pause time drops significantly because only a fraction of the old generation is reclaimed at a time. Splitting objects into cars, however, introduces some complexity to track inter-car references, for example.

2.5 Memory Management Scalability Limitations in the JVM

As discussed in the previous sections, the task of managing memory in a JVM is handled by the GC, which is responsible for several tasks: i) prepare memory to serve allocation requests (i.e., for new objects), ii) ensure that all live objects are kept in memory, and iii) collect memory used by objects that are no longer alive (garbage). The GC is therefore implemented as a set of algorithms that hide most memory management issues from programmers. The use of automatic memory management via GC was a choice taken right from the beginning [54], and it is not in the scope of this work neither to motivate the use of GC, nor to present its advantages or disadvantages regarding explicit memory management (for this discussion, please refer to Jones [67]).

Besides the clear advantages of using a GC, there are some limitations imposed by this component. In practice, GC implementations in the JVM fail to vertically scale memory, i.e., dynamically allow the memory usage to grow and shrink as the application needs. In the rest of this section, the two main reasons for this lack of scalability are discussed.

2.5.1 Reserved vs Committed vs Used Memory

Users can only define the heap size limit at launch time and, during runtime, an application is assured to have a fixed memory area to place application objects (the heap). An 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 may decide to 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 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 (this memory is referred to as `reserved memory`). 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.

2.5.2 GC Data Structures

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 maintains track of references that cross heap sections.

These internal data structures are setup 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, if an application needs more memory than what was defined at launch time, the only solution to change the heap size limit is to re-start the JVM, incurring into a significant application downtime.

To summarize, in this section we studied the main two reasons why current JVM implementations cannot easily scale memory vertically, i.e., increase or decrease the memory available to the application. The main two reasons are: i) heap resizing operations can only occur during a GC cycle (which can take a long time to be triggered, and ii) GC internal data structures cannot be easily resized to handle more memory than what was defined at launch time. Currently the only solution to vertically scale the memory of an application running on top of a JVM, is to restarts it with new memory limits, thus incurring into a significant application downtime.

2.6 Summary

This chapter strives to provide the necessary background to follow the next chapters in this document. In particular, we defined Big Data environments and their components, and studied how they stress memory management algorithms in current runtime systems. It was also possible to conclude that most Big Data platforms rely on multiple worker and/or data nodes to function, reason why most Big Data platforms need to cope node churn (nodes coming and going). A JVM architectural description was also presented before discussing memory management concepts and classic algorithms. The chapter closed with an analysis of the main reasons why current JVMs cannot vertically scale memory.

Chapter 3

Related Work

This chapter analyzes recent works that try to solve problems similar to the ones described in Chapter 1. Thus, the goal of this chapter is to identify differences and drawbacks in current approaches, further motivating the search for better, improved solutions.

We start this chapter by analyzing VM migration algorithms (see Section 3.1). These algorithms present solutions for Problem 1 (migration/replication). After looking into VM migration algorithms, we move into exploring solutions for GC in Big Data applications. These algorithms strive to solve Problem 2 (see Section 3.2). We further divide this GC study into algorithms that are mostly optimized towards throughput or latency (also including some solutions that use object life time profilers to help pretenuring objects). Finally, we present state of the art for resource scalability algorithms in virtualized environments and heap resizing algorithms (see Section 3.3). These algorithms represent previous attempts to solve Problem 3 that emphasizes the need efficiently manage resources and adapt the runtime's resources usage according to the application needs.

Within this chapter, each section closes with an analysis and comparison of previously proposed algorithms. This chapter finishes with a summary of the main conclusions of the related work analysis (see Section 3.4).

3.1 VM Migration Algorithms for Big Data Environments

In this section, we describe current migration algorithms. We start by describing general migration approaches: pre-copy and post-copy, followed by an analysis of the most recent and relevant migration algorithms. We close this section by presenting a taxonomy of the most relevant and recent VM migration algorithms.

Please note that when we refer to VM migration, we are considering both the migration of

Java Virtual Machines (JVMs), containers [8], and the migration of system virtual machines (e.g. Xen-based [9] or similar). When a JVM migration occurs, only a process is migrated to a destination site. On the other hand, a system-VM migration includes the migration of all processes, and the operating system kernel. Please also note that ALMA (the proposed JVM migration algorithm) uses a JVM migration approach.

Solutions to migrate VMs (JVMs or system-VMs) 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) whether solutions use or not optimization techniques (note that multiple optimizations can be used at the same time). In this section, we discuss the motivation behind and differences between pre and post-copy approaches. In the next sections we see that many different solutions optimize VM migration along different aspects.

Pre-Copy

Pre-copy was firstly introduced by Theimer et al. [114] and it is the most common technique to migrate a VM. Using this approach, the bulk of the VM's memory pages is transferred to the destination site while the VM is still running (on the source site). While memory pages are being transferred, changes being made to those pages are being tracked. When the transference of the bulk of the VM's memory is complete, the VM stops executing on the source site and all the pages modified during the transference process are re-transferred. After this step, the VM resumes execution at the destination site.

In the work by Clark et al. [29], the authors propose an improvement over this method which consists on the identification of a small writable working set (WWS). Using this technique, instead of using only two rounds (transferring the bulk of VM pages and then all the modified pages), the migration engine can use an arbitrary number of rounds (during which changed memory positions are transferred to the destination site) in order to identify a small WWS (this process is limited to a maximum number of rounds). In other words, the VM is stopped only when the number of modified pages is small, resulting in a fast transference of modified pages, leading to a very short VM downtime.

Post-Copy

Post-copy (firstly proposed by Zaya et al. [130]) presents a different approach, in which the executing VM is migrated with the minimum number of memory pages possible, and resumes execution at the destination site (while most of the memory pages are still at the source site). The remaining VM pages (at the source site) are transmitted lazily, only when requested at the

destination site. In other words, if an application tries to access some memory page that is not on the destination site yet, a request is issued and the memory page is transmitted before the VM can access the page's content. Although this leads to significant memory latency right after the VM migration (i.e., many memory accesses will trigger page transferences), pages are transmitted only if needed. This means that unused pages may be never transmitted and that less stress is put on the network infrastructure since each page is transmitted at most once (compared to pre-copy approaches in which pages can be transmitted multiple times).

3.1.1 VM Migration Algorithms

We now use this section to discuss the most recent and relevant works in the literature regarding VM migration. We are aware of many works [18, 129, 109, 84, 98, 101, 19, 44] that, despite their relevance, are not discussed because, overtime, the systems described below improved the ideas and algorithms described in those previous works. We conclude this section with a taxonomy comparing each algorithm according different metrics.

Hines et al. [59]

Hines et al. [59] advocate that pre-copy (which is the most common approach for migrating VMs) is good for read workloads but bad for write workloads because the pages will get dirty over and over again, and the working set can be a moving target.

The authors propose the use of post-copy to reduce downtime and to guarantee that each page is transmitted only once. Using this approach (post-copy), the VM starts executing at the destination site before most memory pages are transmitted. Only when an application tries to access some memory position belonging to a page that has not been transfered yet, a fault is triggered and the page is transferred.

Besides the basic post-copy algorithm, the authors also propose two optimizations: i) adaptive pre-caching, and ii) dynamic self-ballooning. Using the first (adaptive pre-caching), the migration engine (in the destination site) pro-actively pre-fetches pages from memory (in the source site) in order to hide the latency problem of fetching pages from a remote site. The second optimization is used to reduce the actual amount of memory used by the application at the source site, and therefore reduce the number of pages that would be considered for transferences between source and destination sites. This is accomplished by using dynamic self-ballooning which gives free pages back hypervisor.

The proposed algorithm is implemented for Xen [9] and, through the evaluation of their implementation, the authors are able to demonstrate that post-copy brings benefits mainly for

throughput oriented workloads (those that do not require short pauses times, which are difficult to cope if too many page faults occur at the same time).

Vogt et al. [120]

Vogt et al. [120] present an alternative to either pre-copy or post-copy by combining both techniques in a single algorithm. The proposed work is specially optimized for high frequency checkpointing (for example, checkpointing the application at each client request) which can go below one millisecond between checkpoints, and tries to overcome the memory tracing overheads of incremental checkpointing (i.e., creating checkpoints that contain only modified memory positions).

The authors present a technique called Speculative Memory Checkpointing. This technique minimizes memory tracing overheads by eagerly copying the hot (frequently changing) memory pages while lazily tracing and copying at first modification time only cold (infrequently changed) memory pages. In order to find the optimal trade-off between the set of hot memory pages and cold memory pages, several working set estimation algorithms can be used [70].

The proposed algorithm is implemented as a loadable kernel module for Linux. Experimental results show that it is possible to reduce performance overhead associated to high frequency checkpointing while maintaining a low memory overhead (compared to previous solutions).

Jin et al. [64]

Jin et al. [64] point out that most migration algorithms use pre-copy techniques that tend to exhaust the network bandwidth due to many page transferences necessary to transfer all the VM state and all subsequent iterations (to reduce the working set size) until the control is passed to the destination site.

The authors propose a solution for this problem that uses memory compression. The idea is that multi-core machines (which are very common) usually have spare resources that are idle or under utilized most of the time. Using these spare resources, it is possible to compress memory pages before being sent to the destination site and decompress them at the destination site. The authors also propose a compression algorithm which strives to maintain the compression rate higher than the available network bandwidth. In other words, the algorithm only compresses if it is worth, i.e., if compressing and transferring is faster than just transferring.

The proposed algorithm is implemented for the Xen and experiments conducted show that it is possible to reduce application downtime by up to 27%, the total migration time by up to 32%, and the total transferred data by up to 69%.

Deshpande et al. [36]

Deshpande et al. [36] address the problem of migrating groups of VMs at the same time. This is a frequent technique used in order to handle resource re-allocation for peak workloads, imminent failures, cluster maintenance, or powering down several physical nodes. The problem with migrating large groups of VMs at the same time is that network links are easily exhausted (even in 10 Gb/s networks), preventing applications from progressing due to the lack of network bandwidth.

The proposed solution is inspired in the fact that VMs within the same cluster will often share the same applications, libraries, and other tools so that many memory pages will be identical in multiple VMs. Therefore, it is possible to only send one copy of such pages to the destination site even if multiple VMs are scheduled to be re-located at the same node. The authors introduce an approach called gang migration using global deduplication. During the normal execution of applications, a duplicate tracking mechanism is necessary to track identical pages across multiple VMs.

This technique is implemented in KVM [74] and the authors conducted experiments which revealed improvements of up to 42% for the total migration time, and reduction of the used network bandwidth by up to 65% when compared to the default migration strategy used in KVM.

Knauth et al. [75]

Knauth et al. [75] propose a new idea for reducing the network bandwidth traffic overhead related to VM migrations. The authors, use the insight that VMs do not migrate randomly between hosts, but instead follow a pattern which reveals the fact that only a small set of servers is ever visited by the same VM.

To decrease the migration traffic and time, the authors propose that each source migration site stores a checkpoint of the outgoing VM locally. As the probability is high that the VM will return to the source site at some point in the future, the incoming migration can be bootstrapped with the old checkpoint.

The proposed algorithm is implemented in KVM and, through experiments based on real-world traces, the authors were able to reduce the migration traffic up to 75% compared to a full system migration.

Kawachiya et al. [72]

The problem of slow JVM startup is investigated by Kawachiya et al. [72]. The authors point out that, in many situations, administrators need to replicate or clone JVMs running server applications (for example, web servers) to accommodate a new workload or to replace failed nodes. However, despite having other JVMs running the same application in the same physical node or virtual machine, creating a new one still encompasses a lengthy process (which can take dozens of seconds for large applications with thousands of classes to load and compile).

In order to solve this problem, the authors suggest cloning already running JVMs (instead of starting them from scratch). In other words, JVMs are migrated (or cloned) to provide fast startup of the applications running on these JVMs. This obviously only makes sense if the same application runs and if the workload is the same or very similar. Applications can also control their own clones by using a cloning API (which is integrated in the JVM).

This algorithm is implemented in a production JVM, the IBM J9 JVM.[1] Evaluation experiments show that cloning a JVM instead of creating a new one can speed up the bootstrap process by 4 to 170 times faster.

Hou et al. [60]

Migration of write intensive applications can be a problem for pre-copy approaches since the working set of an application is constantly changing (i.e., it is a moving target). This leads to a situation in which it is difficult to isolate a small working set that would provide a small application downtime.

Hou et al. [60] explore the idea of using application assistance to decrease the size of the working set is proposed. In this work, the authors create a stub that is placed between Xen, the migration engine, and the running application (that runs inside a JVM). Using information provided by the stub (regarding the state of the application), the migration tool is able to skip the transference of pages that are no longer necessary for the application.

In practice, this algorithm uses a pre-copy approach that, after performing a full transference of all the VM pages (note that this algorithm is targeted to migrate JVMs but it migrates whole system-VMs), triggers a minor garbage collection. After this collection, all reachable objects are promoted to the old generation and therefore the young generation is empty (i.e., with no live objects). With the application still stopped from the minor collection, the stub informs the migration tool that the JVM is ready to be migrated, and that all pages belonging to a particular memory range (where the young generation is located) should not be transferred.

[1] IBM J9 is a JVM developed by IBM.

The system is implemented using two components: i) a kernel module that moderates the communication between a modified version of Xen and the JVM, and ii) a JVM agent (a pluggable component that run attached to the JVM and has access to its internal state) that is able to inspect the application state and therefore identify the memory pages with no live objects. Evaluation results were obtained using the SPECjvm2008[106] benchmark suite. It was possible to conclude that the application downtime can be reduced by up to 90% when compared to vanilla Xen migration.

Li et al. [81]

Current cluster schedulers typically use preemption to coordinate resource sharing, achieve fairness, and satisfy service level objectives. However, the current (or mostly used) mechanism to preempt a process is by killing it and restarting it latter. This obviously causes significant resource waste, and delays the response time of long running jobs.

Li et al. [81] propose a solution through a checkpoint-based preemption algorithm, which does not discard the progress of applications when these are to be preempted. Adaptive preemption policies are also proposed to mitigate the suspend-resume overheads. The adaptive policies dynamically select victim tasks and the appropriate preemption mechanisms (e.g., kill vs. suspend, local vs. remote restore) according to the progress of each task and its suspend-resume overhead.

This algorithm is implemented using CRIU [71], HDFS, and PMFS [46]. The implementation is integrated with Hadoop Yarn [119]. Experiment results show that this approach achieves up to a 67% reduction in resource wastage, 30% improvement in overall job response times, and 34% reduction in energy consumption over the current YARN scheduler.

Gioiosa et al. [52]

Gioiosa et al. [52] idealize a solution that automatically detects, diagnoses, and migrate applications in order to solve software and/or hardware problems without changing existing applications. According to authors, the solution must also be aware of other running systems in remote sites that need to be coordinated with the system that is being migrated (for example, a set of nodes working on a single MPI problem).

In order achieve such a system, the authors propose a solution with two main components: i) operating system support for checkpoint, migration, and restore operations for unchanged applications (i.e., application must not be aware of the migration algorithm), and ii) global orchestration to reach consistent recovery across a large number of nodes.

Algorithm	Pre/Post Copy	Target	Type	Reachable Data
Hines et al. [59]	post-copy	system-VM	migration	no
Vogt et al. [120]	mixed	process	checkpoint	no
Jin et al. [64]	pre-copy	system-VM	migration	no
Deshpande et al. [36]	pre-copy	system-VM	migration	no
Knauth et al. [75]	pre-copy	system-VM	checkpoint	no
Kawachiya et al. [72]	post-copy	process	checkpoint	yes
Hou et al. [60]	pre-copy	system-VM	migration	yes
Li et al. [81]	pre-copy	system-VM	migration	no
Gioiosa et al. [52]	pre-copy	system-VM	migration	no
CRIU [71]	pre-copy	process	checkpoint	no

Table 3.1: Taxonomy of VM Migration Algorithms

The system is implemented inside the Linux kernel, providing transparent checkpoint, migration, and restore to applications. Buffering co-scheduling [100] is a technique used to synchronize inter-node communication periodically (so that restoring can be done to a previous global consistent state). Experimental results show that incremental checkpointing with few seconds between each checkpoint can increase the runtime about 10%.

CRIU [71]

CRIU [71] 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, not in the Linux kernel.

CRIU supports snapshotting 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 [45] live migration.

3.1.2 VM Migration Algorithms Comparison

Having described each algorithm in separate, we now summarize all algorithms in Table 3.1. The table describes each algorithm along several perspectives: i) if it uses a pre-copy or post-copy approach (or even mixed); ii) if the migration algorithm is targeted to system-VM migration or process migration; iii) if the migration algorithm only handles the creation of snapshots/checkpoints or if it also handles the migration and restoration of the snapshots at the remote site; and iv) if the algorithm avoids unreachable data (garbage) when creating snapshots/checkpoints.

From Table 3.1, it is possible to draw some conclusions:

- pre-copy is the most popular migration technique. Most systems use pre-copy, which uses more network bandwidth than post-copy. The benefit from using pre-copy is the short application downtime, which is the most important metric for many applications, typically latency sensitive applications (i.e., applications that interact directly with the end-user);

- no process migration algorithm supports migration between two physical sites. In other words, from all the analyzed algorithms, all system-VM migration engines support migration of system-VMs between sites while not a single process migration engine supports process migration between sites. This simply demonstrates the complexity of migrating a process between two nodes. CRIU supports restoring a snapshot of a process in a remote site, however it does not transfers the snapshot automatically and therefore, the user is responsible for performing the transfer;

- only two (out of 10) algorithms do not include garbage when creating snapshots of system-VMs or processes. This is an important improvement over existing migration techniques specially in memory managed runtimes (such as the JVM) in which large heaps would be migrated despite having only most of the memory unused, i.e., with unreachable objects or with no objects.

To conclude our analysis regarding current VM migration algorithms, with Table 3.1, it is possible to conclude that there is no process migration algorithm able to migrate only reachable/usable data. In other words, i) current process migration algorithms are not able to avoid garbage when creating snapshots, and ii) to avoid garbage when creating snapshots, one must use system-VM migration algorithms which force the migration of the whole system (including the OS), which is frequently not necessary.

3.2 Garbage Collection Algorithms for Big Data Environments

In Section 2.4, the two classic approaches to collect garbage (reference counting and tracing) have been addressed. These algorithms, however, show several problems which limit the scalability of today's Big Data platforms.

Starting with reference counting algorithms, there are two main problems. First, these algorithms are not complete and therefore need extra techniques to collect cycles of garbage (such as trial deletion). Trial deletion, comes at a very high cost in terms of computational cost (reducing application's throughput) since it has to simulate the deletion of a possibly large number of objects. The larger the object graph is, the longer trial deletion can take. Second, there

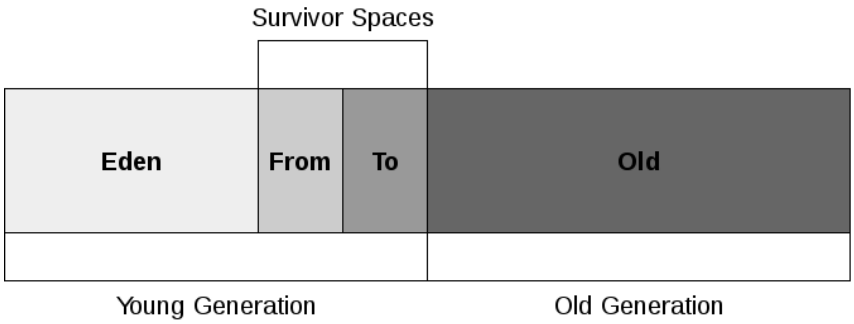

Figure 3.1: CMS Heap Layout

must be write barriers on all reference modification instructions (to account for new and deleted references). This obviously incurs into major application throughput penalties since, after each reference write, the application is stopped and the GC steps in to fix reference counters.

For these two reasons (which impose a severe impact on application throughput), reference counting algorithms are not used in most production JVMs such as the OpenJDK HotSpot JVM (the most widely used JVM implementation).

Tracing algorithms are the most used type of GC algorithms nowadays. Maintained implementations such as the Concurrent Mark Sweep (CMS) and Parallel Scavenge (PS), two widely used production GCs available in HotSpot, combine a set of techniques to optimize the GC process. The main difference between the two collectors is that CMS is concurrent (with the application) when tracing/marking the heap while PS is not concurrent. For this reason, CMS is the most used collector in production environments as it delivers shorter pause times while maintaining good throughput (when compared to PS). Therefore, we focus our analysis in CMS.

CMS, is a generational collector with two generations: young and old (see Figure 3.1). The young generation is further divided into: i) the Eden, where all newly allocated objects are placed; and ii) the survivor spaces, which are used to hold objects that survived at least one collection but are not old enough to be promoted to the old generation. A parallel copy collector periodically (in most implementations, when the Eden space gets full) traverses the young generation and copies objects of a certain age (implementation specific) to either one of the survivor spaces or to the old generation. The old generation uses a parallel, concurrent, and non-moving Mark Sweep collector to reclaim objects residing in the old generation. This collector has been shown to offer acceptable performance (throughput and pause time) for many applications.

However, when it comes to most Big Data platforms, with massive amounts of objects in memory and with high throughput and pause time requirements to cope with, CMS can be a

limiting factor mainly because of three major problems.

First, tracing algorithms (and therefore CMS as well), have to traverse the whole heap to identify garbage. This becomes a problem if the size of the heap grows to very large sizes (hundreds or even thousands of GBs). In such scenarios, the process of tracing the heap (which is concurrent with application threads) can take so long that eventually memory is exhausted and therefore a full collection (that collects the whole heap) is triggered. These collection cycles can take dozens or even hundreds of seconds to collect all objects in memory. This obviously has a severe impact on throughput and pause time for the running applications.

Secondly, the other problem with CMS is directly related to the memory profiles analyzed in Sections 2.1.1 and 2.1.2. We have seen that both processing and storage platforms can keep many live objects in memory: working sets in the case of processing platforms, and caches in the case of storage platforms. Please also remember that all these objects (belonging to working sets and caches) are allocated in the young generation which, when full, is collected. During this process, all live objects are copied to the old generation. This copying process is slow and is limited to the memory bandwidth available on the hardware. Therefore, and since processing platforms keep their active working sets alive (usually one per task/core), and storage platforms keep their caches alive (usually one per table/database), many live objects will be copied within the heap, leading to frequent and length full collections (reducing an application throughput and increasing an application pause time). In other words, although the fact that the well-established assumption that most objects die young still holds in most Big Data platforms, the pause time impact inflicted by the small percentage of objects that live longer is not negligible [96].

To further aggravate this problem, no naive solution is applicable. Increasing the size of the young generation reduces the number of times the young generation is full and therefore reduces the number of collections, increasing the application throughput. However, this increases the average pause time for young generation collections. On the other hand, reducing the size of the young generation increases the number of young generation collections (decreasing the application throughput) but reduces the application pause time. Finally, adding more cores will not help because the object copy process is bound to memory bandwidth.

The third problem is fragmentation in the old generation. As objects with longer life cycles[2] live, objects with shorter life cycles (but already in the old generation) will become unreachable. This results in a highly fragmented old generation which leads to decreased locality, and can

[2]An object's life cycle is a term used hereafter to refer to the moment of creation and collection of a particular object or set of objects. Thus, objects with similar life cycles are created and collected approximately at the same time.

even lead to situations where no more memory can be allocated (although there is enough free space) because of fragmentation.

To conclude, current collectors provided by production JVMs still present scalability challenges that need to be addressed. For this reason, several solutions have been published to try to alleviate these problems. In the next sections, we look into several relevant solutions. We divide our analysis into throughput oriented and pause time oriented solutions because most of these solutions are focused on improving or have the largest impact on one of these metrics.

3.2.1 Throughput Oriented Memory Management

Several improvements have been proposed for reducing the negative impact that GC has on applications' throughput. In this section, we study some of the most recent and relevant GC solutions that try to increase the application throughput by removing the overhead introduced by automatic memory management (i.e., GC).

Gog et al. [53]

Gog et al. [53] propose Broom, a memory management algorithm that uses region-based memory management as a way to reduce the cost of managing massive amounts of objects usually created by Big Data processing platforms.

The authors want to take advantage of the fact that many objects created by processing platforms (Naiad, for this specific work) have very similar life cycles. By knowing this, Broom enables platform developers to group all these objects whose life cycles are similar in separate regions. These regions could be easily collected (including all the objects within) whenever the objects within these regions are not necessary anymore. In other words and relating to the concepts introduced in Section 2.1.1, Broom stores objects of different working sets in different regions; after a task is complete, the working set is discarded and the region is freed (knowing that all objects within will not be used again).

Three types of regions are proposed: transferable regions, task-scoped regions, and temporary regions. Transferable regions are used to store objects that persist across tasks and can be used by different tasks across time. Task-scoped regions are meant to store objects belonging to a single task. Finally, temporary regions are used to store temporary objects; these objects cannot persist across method boundaries.

To avoid complex reference management between regions, Broom does not allow references from: i) objects inside temporary regions to objects inside task-scoped; ii) objects inside temporary regions to objects inside transferable regions, and iii) objects inside task-scoped

regions to transferable regions. This way, objects that live for longer periods of time never reference objects with smaller life times and therefore, no region is kept alive because of other region.

Using Broom prototype implemented for Mono (a Common Language Runtime for Linux), the authors were able to reduce the task runtime of Naiad for up to 34%.

Despite the positive results, Broom presents some limitations: i) the programmer must have a very clear understanding of the objects' life cycles in order to be able to group them properly into regions; ii) this is even aggravated by the fact that inter-object references are limited (objects from temporary regions cannot reference task-scoped regions, for example); iii) Broom is only a prototype used for Naiad, i.e., it only works with Naiad, meaning that it cannot be used with other Big Data platforms.

Nguyen et al. [95]

Nguyen et al. [95] propose FACADE, a compiler framework for Big Data platforms. The proposed system takes as input any Big Data platform bytecode and modifies the code to use native memory (off-heap) instead of the GC-managed memory (on-heap). Native memory or off-heap is a way to access memory that is not managed by the GC. When using native memory, the programmer is responsible for allocating and deallocating memory (much like in a C/C++ application). The idea behind FACADE is that all the native memory code (potentially hard to code and to debug) is automatically generated and replaces regular Java code.

Using the transformed bytecode, the platform is able to reduce the number of objects in the GC-managed heap memory, thus reducing the GC effort to keep these objects in the managed heap, leading to an increase in the application throughput. Relating to the concepts explained in Section 2.1.1, FACADE is pushing objects belonging to working sets to native memory (i.e., out of the reach of the GC).

The problem of avoiding GC by pushing objects into off-heap is that the programmer must explicitly collect all memory. In other words, FACADE must be able to collect all objects that are allocated in native memory. In order to solve this problem, FACADE requires the programmer to specify when a new working set is created and when a working set can be collected (note that FACADE does not allow the existence of multiple separate working sets at a time). Therefore, this system is mostly appropriate for iteration-based processing platforms, whose working sets are discarded by the end of each task/iteration.

The authors successfully used FACADE to transform the bytecode of seven Big Data applications across three Big Data platforms: GraphChi [78], Hyracks [15], and GPS [105]. Results

showed that the execution time can be reduced by up to 48 times.

The main drawback presented by this solution is its limitation regarding the range of workloads that can be used. Since FACADE only allows one working set (per-thread) at a time, it does not support non-iterative workloads such as the ones typically associated with storage platforms. In storage platforms, working sets (caches) are not bound to a single thread (while on processing platforms, processing tasks usually are) thus making it very difficult to use FACADE. Another related problem is the way FACADE requires programmers to identify when working sets start and finish. Between these two code locations, FACADE intercepts all allocations and places them in off-heap, meaning that programmers must remove all non-data objects from within these boundaries. A final comment on FACADE's evaluation is that it is done using the Parallel Scavenge GC, an obsolete and unrealistic GC for many Big Data platforms. Current GCs used in realistic OpenJDK production settings are usually CMS or G1 (described next).

Lu et al. [85]

Lu et al. [85] propose Deca, an extended/modified version of Spark which tries to reduce the GC overhead present in Spark because of its massive creation of objects with very different life times (i.e., some objects have a very short life time while others live for a long period of time). The authors propose a life time-based memory management so that objects are grouped according to their estimated life time.

Using this approach, objects created by the platform (which will potentially live for a long period of time) are serialized into large buffers thus avoiding continuous GC marking and tracing. By keeping the bulk of the data maintained in memory (by Spark) inside large buffers, Deca is able to reduce the GC marking and tracing overhead, and therefore it is able to increase the platform throughput.

As with previous systems (such as FACADE), one problem of maintaining serialized versions of objects is how to keep their memory consistent while efficiently reading and writing to it. Deca solves this problem by pre-allocating large arrays where objects will fit into. To determine the size of these arrays, Deca estimates the size of each data object (actually it uses an upper bound of the size).

In practice, the solutions proposed by Deca and FACADE are similar. Despite the fact that the first only works for Spark and the second works for any iterative workflow platform, both of them try to hide massive amounts of data objects from the GC to avoid the GC overhead associated with keeping these objects in memory (namely the tracing overhead). Relating to the concepts introduced in Sections 2.1.1 and 2.1.2, Deca is pushing the working sets and

Spark's intermediate data (similar to the caches present in storage platforms) into large buffers, away from the collector.

The authors were able to improve Spark throughput by reducing its execution time by up to 42 times (compared to normal execution, using the default GC), using workloads such as Connected Components, Page Rank, Word Count, among others.

Deca is, however, specific to a single processing system, Spark. In other words Deca cannot be used in other platforms. Worse, the technique used to modify Spark (allocating objects in large arrays) is often unpractical as object allocations happen in so many code locations (making it harder to change from heap allocations into array allocations), and therefore requiring a major rewriting the platform.

Gidra et al. [51]

NumaGiC (proposed by Gidra et al. [51]) presents several developments to improve GC performance in cache-coherent Non-Uniform Memory Access (NUMA) environments. The authors propose several mechanisms to reduce the amount of inter-NUMA node reference tracing performed by GC threads. By improving reference tracing locality (i.e., only trace references local to the current NUMA node where the GC thread runs), NumaGiC is able to improve applications' throughput.

With this collector, objects are placed in specific NUMA nodes not only upon allocation but also upon copying (after a collection). The most appropriate NUMA node to place an object is determined using several policies:

- new objects are placed in the same NUMA node where the mutator thread that creates the object is running;
- the roots of a GC thread are chosen to be located mostly on the NUMA node where the GC thread is running;
- objects that survive a young collection are copied to the same NUMA node where the GC thread (that handles the object copying) is running;
- upon heap compaction, NumaGiC tries to maintain objects in the same NUMA node.

With these policies, it is still possible to end up with an unbalanced distribution of objects, i.e., some NUMA nodes can end up having most objects allocated in it. To solve this problem, GC threads running on different NUMA nodes steal work from other GC threads. If a GC thread finds a reference to an object residing in a remote NUMA node, it notifies the remote GC thread (running on the corresponding NUMA node) to process that object.

NumaGiC is implemented on the OpenJDK HotSpot 7 JVM as an extension of the existing Parallel Scavenge GC (which is similar to CMS but it is not concurrent). The authors compared their new collector with NAPS [50] (NUMA-aware Parallel Scavenge) using platforms such as Spark and Neo4J, and improved the throughput of those platforms by up to 45%. Nevertheless, it would be very interesting to confirm that the benefits obtained with Parallel Scavenge can also be obtained with concurrent marking GC (which is the most realistic setup nowadays) such as CMS.

Cohen et al. [31]

Cohen et al. [31] propose the Data Structure Aware GC (DSA for short), a collector that tries to benefit from the fact that particular objects are inside a data structure to improve collector's performance and therefore, alleviate the GC overhead on the platform's throughput. The motivation behind DSA is that: i) there are many Big Data platforms which are data structure oriented (mainly storage platforms), and ii) a collector able to distinguish objects that are inside a data structure (and therefore are alive) would avoid handling (tracing for example) these objects in the hope that a large portion of the overhead caused by the collector would be eliminated.

The programmer is required to explicitly tell DSA: i) which classes are part of a data structure, and ii) when objects belonging to a data structure should be collected. If the programmer fails to report the deletion of an object or reports false information, the correctness of the collector is not compromised; the only consequence is a degradation in the performance of DSA to collect such objects.

Objects belonging to a data structure are allocated in a separate heap area, away from other regular objects. According to the authors, this also provides locality for the objects inside the data structure (that are collocated in the same heap area). From the collector point of view, objects belonging to a data structure are considered as root objects. Tracing is improved by having most data structure objects in the same heap area (benefits from locality).

Relating this work with concepts introduced in Section 2.1.2, DSA is pushing caches into a separate heap to improve locality and therefore improve the performance of the collector and platform.

DSA is implemented on JikesRVM [4] (a research JVM) and it was tested with KittyCache, SPECjbb2005, and HSQLDB. DSA improved (with regards to using the default collector) the throughput up to 20% using KittyCache, 6% using SPECjbb2005 and 32% using HSQLDB.

However, DSA is implemented in a research JVM and not a production JVM, which difficults comparing its results with other approaches available. Furthermore, DSA requires the

programmer to inform the JVM: i) of all the classes that should go into a separate space (data structure space), and ii) whenever an object inside a data structure is removed. This requires a lot of effort from the programmer. An additional problem is that some objects belonging to a data structure class might never go into a data structure (i.e., can be temporary objects). This breaks the objective of the solution and DSA has no apparent way of preventing this.

3.2.2 Pause Time Oriented Memory Management

Having discussed several throughput oriented systems, we now present GC solutions whose main goal is to reduce the application latency introduced by automatic memory management (i.e., GC).

Maas et al. [88]

Maas et al. [88] propose Taurus, a holistic language runtime system for coordinating Big Data platforms running across multiple physical nodes. The authors point out that these platforms run distributed on multiple physical nodes, and that each node uses a managed runtime system such as a JVM to run a Big Data platform on top of it. However, each runtime is not aware of the existence of others working for the same platform (and possibly running the same application).

This lack of communication between runtime systems leads to individual runtime-level decisions that optimize the performance of the local runtime but that are not coordinated, leading to undesirable scenarios if the whole platform performance is not considered. For example, if a JVM starts a new collection while other JVMs are already running collections, although it might be beneficial for the performance of the local node to start a collection, it might lead to significant platform level latencies because many JVMs are paused for collection at the same time.

Taurus solves this problem by presenting a holistic runtime system that makes platform-level decisions for the entire/global (cluster-wise) platform. Therefore, and reusing the previous example, using Taurus, JVMs are periodically requested (by Taurus) to start a collection at different times therefore minimizing the number of JVMs paused for collection at any time.

Using Taurus, application developers can supply policies (written in a special DSL) to instruct Taurus how to coordinate runtime-level events such as GC. This solution is based on the OpenJDK HotSpot JVM and is implemented as a JVM drop-in replacement. The authors were able to reduce Cassandra read latency by around 50% and write latency by 75% (both results for the 99.99 percentile, compared with normal execution using an unmodified JVM).

The obvious limitation of Taurus is the assumption that there are always enough spare

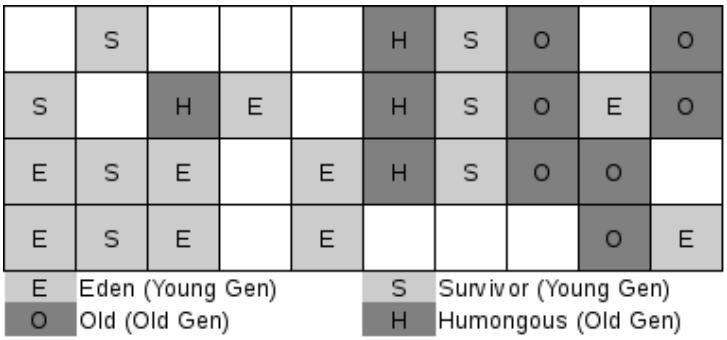

Figure 3.2: Garbage First GC Heap (each square represents a region)

resources to replace the nodes that need to go under maintenance (for example, running a GC cycle). This is not obvious if multiple nodes require maintenance at the same time or if maintenance takes too long. In such situations, for example, during fast workload changes, the number of nodes that need to go under maintenance can easily go over the number of spare resources, resulting in high application latencies.

Detlefs et al. [37]

Garbage First (proposed by Detlefs et al. [37]), G1 for short, is the most recent collector available in the OpenJDK HotSpot JVM, being the current default collector in OpenJDK 9. G1 represents an evolution regarding CMS with the goal of being able to reduce applications' pause times while keeping an acceptable throughput. Its main idea is to divide the heap into small regions that can be collected independently (if needed) in order to maximize the amount of collected garbage while staying below the max acceptable pause time. By doing so, G1 also eliminates the need for full collections (which were known to lead to unacceptably long pause times in CMS and PS).

As with CMS, G1 is generational (i.e., the heap is divided into young and old generations) and, therefore, each heap region can be either in the young generation or in the old generation. Young generation regions are further divided into Eden (space where all objects are allocated) and Survivor (space where objects that survived at least one collection but are not old enough to be promoted to the old generation live). Old generation regions are divided into Old (contain objects that survived several collections, and therefore are considered old) and Humongous (contains large objects that are allocated directly in the old generation). Figure 3.2 presents a graphical representation of a possible G1 heap.

G1 has three types of GC cycles:

- minor collections, only regions belonging to the young generation are collected;

- mixed collections, all regions belonging to the young generation are collected and some regions from the old generations are also collected (this process is described further below);

- full collections, all regions belonging to both generations are collected.

One of the main benefits of having the heap divided into regions is the possibility to perform mixed collections, where the collector selectively collects regions from the old generation. This keeps the heap from being fragmented and without free space.

Regions belonging to the old generation are selected for a mixed collection according to their amount of live data. Regions with more unreachable data will be the first to be selected to be included in a mixed collection (hence the name Garbage First). This serves two purposes: i) collecting regions with less live data is faster than collecting regions with more live data (since we do not have to copy so many objects to other regions), thus improving the performance of the collector; ii) since the collector has a maximum acceptable pause time for a collection (which is a user-defined constant), regions which are faster to collect are also easier to collect while still being able to respect the limit pause time.

G1 relies on periodic concurrent marking cycles (a concurrent marking cycle traverses the heap marking each live object) to estimate the amount of live objects in each region. This information, together with other statistics built over time regarding, for example, previous collections, is used to estimate the time needed to collect each region. With this information, the collector identifies a set of regions to collect that maximizes the amount of garbage collected but still does not exceed the maximum desirable pause time.

The authors show that G1 is able to respect a soft real-time pause time goal and that it was not possible to obtain such pause times with CMS. Two well known benchmarks were used: telco and SPECjbb.[3]

Although representing a major improvement regarding previous production GCs (Parallel Scavenge and CMS), G1 suffers from very long GC pauses when Big Data platforms create large portions of objects with long life cycles, for example: i) the creation of a working set (when a task starts) in processing platforms, and ii) the creation of caches in storage platforms. In both situations, GC pauses are very long due to object copying between heap spaces and go over the maximum desirable pause time set by the user.

[3]The telco benchmark can be found at http://speleotrove.com/decimal/telco.html. SPECjbb benchmark suite can be found at https://www.spec.org/jbb2015/.

Jump et al. [69]

Jump et al. [69] propose an object life time profiler that samples object allocations in order to be able to estimate if an object will live for a long time or not. According to the authors, this information could be of great importance in current generation GC design as all GCs need to either estimate or simply assume the estimated life time of objects. Since most GC assume that objects will die young, it makes sense to have an object life time profiler that accuratelly estimates how long an object will live.

The proposed profiler samples allocations by every N allocated bytes (where N is a configurable parameter). For each selected object, the profiler will insert additional information to the allocated object (i.e., increasing the overall size of the object). This information is latter used to produce statistics regarding the estimated life time of objects allocated through the same allocation site.

Using this technique, Jump et al. are able to implement a pretenuring GC algorithm that avoids the cost of promoting objects that are known in advance to live for a long time. The system is implemented on top of the JikesRVM[4], an experimental/academic JVM written in Java. Based on the results reported using several benchmark suites, with a very small throughput overhead, accurate life time estimates are possible, thus improving the performance of the collector and reducing pause times.

Tene et al. [113]

The Continuously Concurrent Compacting Collector, C4 for short, proposed by Tene et al. [113], is a collector developed by Azul Systems.[4] This is a tracing and generational collector such as G1 and CMS are, but it is also distinct from previous collectors by supporting concurrent compaction (which is not supported neither by G1 nor by CMS). In other words, C4 does not require stop-the-world pauses to collect garbage (note that G1 and CMS do require stop-the world pauses, during which all application threads are stopped, to collect garbage).

The C4 garbage collection algorithm is both concurrent (GC threads work while application threads are still active and changing the object graph) and collaborative (application threads can help GC threads doing some work, if needed). The GC algorithm relies on three phases:

- Marking, during this phase, GC threads traverse the object graph, marking each live object. This phase is very similar to concurrent tracing already present in G1 and CMS;

[4] Azul Systems is a private company dedicated to building runtime systems capable of executing the same applications that run on a JVM. It is available at www.azul.com.

- Relocation, where live objects are moved to a free space (also known as compaction). During this phase, all live objects, marked in the marking phase, are relocated. This process is concurrent (GC threads work concurrently with application threads) and collaborative (in the sense that application threads help moving an object if they try to access it before the object is in its new location).
- Remapping is the final phase where references still pointing to objects' old locations, where an object was moved from, are updated. This phase is also concurrent and collaborative (application threads trying to access an object that was moved out will get the new address of the object and automatically update the reference to point to its new address).

By relying on these three phases, which are concurrent and mostly collaborative, application threads are stopped for a very short period of time to correct some reference or help moving an object, but there will never be a stop-the-world pause that stops all application threads for a long period of time. This, however, comes at the price of heavily relying on read barrier handling that reduces the overall application throughput (as shown in Section 6.3.6).

To evaluate C4, the authors used several benchmarks from the SPECjbb2005 benchmark suite. Using transactional oriented workloads (from SPECjbb2005 benchmarks) C4 showed to reduce the worst case pause time by up to two orders of magnitude when compared to CMS.

C4's latency benefits eventually come at the cost of reduced overall throughput or increased resource utilization due to the extreme use of barrier/trap handling. Furthermore, long GC pauses can still occur if the memory allocation rate is above rate at which the concurrent collector can free memory (for example, during workload shifts).

Clifford et al. [30]

Clifford et al. present a the problem of JavaScript execution in web browsers. In such scenario (web browsers) web sites are becoming more and more complex, requiring more memory to execute. This obviously leads to more memory management pressure, which has to be handled by the V8 runtime system (that provides automatic memory management).

In their work, the authors present a solution based on pretenuring whose goal is to allocate objects, that are expected to live for a long time, directly in the old generation. This will reduce the cost of promoting objects and therefore, minimize the pause times. This is particularly important in JavaScript runtimes as the interface thread (which controls what the user is able to see in the browser) is blocked while collector is working.

To determine if an object is going to live for a long time, Clifford et al. use an allocation profiler that installs an allocation context (refereed as Memento by the authors) attached to the

Algorithm	Black/White Box	Developer Effort	Target Platform	Main Goal
Broom [53]	white	high	processing (Naiad)	throughput
FACADE [95]	black	low	iterative processing	throughput
Deca [85]	black	high	processing (Spark)	throughput
NumaGiC [51]	white	none	processing,storage	throughput
DSA [31]	white	medium	processing,storage	throughput
Taurus [88]	black	low	processing,storage	latency
G1 [37]	white	none	processing,storage	latency
Jump [69]	white	none	processing,storage	latency
C4 [113]	white	none	processing,storage	latency
Memento [30]	white	none	processing,storage	latency

Table 3.2: Taxonomy of Big Data Memory Management Algorithms

newly allocated object. This context is used to trace back an object to its allocation site and is discarded upon the next GC. Using this technique, it is possible to estimate if an object that is allocated through a particular allocation site is likely to survive at least on GC cycle or not. If so, it will be pretenured into the old generation the next time it is allocated.

With regards to the evaluation, the authors presented a number of representative benchmarks that revealed great improvements in the number and duration of GC cycles.

3.2.3 Memory Management Algorithms Comparison

To conclude our study in memory management algorithms, we present a table comparing all presented solutions (see Table 3.2). Table 3.2 is divided into several columns, each of which concerning a different feature of each solution:

- (Black/White) **Box**, a black box algorithm is one that does not interfere with the original GC algorithm itself. In other words, this algorithm does not change the GC although it might produce effects on it (such as alleviate its work). A white box solution is one that changes the original GC implementation for improving it;

- (none/low/medium/high) **Developer Effort**, measures the effort needed to apply the algorithm to an existing Big Data platform and/or to an application (running on top of the platform). If no effort is required, for example in G1, it means that the developer does not have to change the platform/application to take advantage of the benefits offered by, for example, G1;

- (processing/storage) **Target Platform**, the platform type where this algorithm is designed to run into;

- (throughput/latency) **Main Goal**, if the algorithm's main goal is to improve throughput or latency.

From Table 3.2 it is possible to observe that FACADE, Deca, and Taurus provide a black box solution, i.e., these algorithms do not change or replace any GC algorithm, they only alleviate the amount of work given to the GC.

Regarding the developer effort, most algorithms require some developer effort (only NumaGiC, G1, Jump, Memento, and C4 do not need code modifications). These modifications can be seen as a serious drawback since it requires source code access and specialized knowledge that only Big Data platform and application developers might have. Algorithms which require high developer effort (Broom and Deca) can be very difficult to use due to high implementation costs.

Broom, FACADE, and Deca, are optimized only for a specific processing platform or subset of platforms while all other platforms are designed to work with both processing and storage platforms.

Memento and Jump employ an online profiler that runs inside the runtime system and instructs the collector on how to pretenure. However, these systems still present some problems. First, both profilers only decide if the object should be pretenured or not. This means that fragmentation will occur in the old generation if two objects have two different life times but both long enough to trigger the pretenuring code. In addition, both systems insert additional information into the object leading to more memory usage and potentially resulting in premature promotion of objects (i.e., since the young generation gets full faster).

Second, both profilers lack context tracking. In other words, if the same allocation site produces objects with very different life times depending on the caller of the method that contains the allocation site, the profiler will produce wrong information. In Jump et al., authors argue that the JIT compiler will inline many calls and therefore, context will be automatically tracked. However, this seems unlikely to be true in most a Big Data application, whose application stack can grow significantly. In addition, for the applications used in the Evaluation chapter, we detected that this is not true, i.e., the JIT compiler cannot encapsulate enough context to precisely estimate object life times.

Finally, Jump solution is implemented in JikesRVM, a research JVM which can potentially hide some performance trade-offs when compared to industrial JVMs (such as OpenJDK HotSpot).

As described in the previous sections, each algorithm has most positive impact on either throughput or latency. Most of these algorithms have, nevertheless, a positive impact (although

lower) on the other metric as well.

To conclude, we consider both the developer effort and the target platform as the two most important factors when considering each algorithm. Algorithms should have low or no associated developer effort to facilitate its introduction into Big Data platforms and should be as generic as possible so that the algorithm can be applied to a wide range of platforms and workloads. In particular, Broom, FACADE, and Deca are only applicable to a sub-set of processing platforms. NimaGiC is applicable to both processing and storage platforms and with no developer effort but only solves problems related to memory accesses in NUMA processors. DSA requires a considerable amount of developer effort. With regards to latency oriented solutions, all five solutions are applicable to both processing and storage platforms and require low (Taurus) or no (G1, Jump, Memento, C4) user effort. However, none of these solutions are able to reduce (or even eliminate) fragmentation, the main cause for long application pauses (more details on Section 6.3).

3.3 Resource Scalability of Big Data Environments in the Cloud

This section discusses resource scalability of Big Data environments in the cloud, one, if not the most popular approach to deploy Big Data environments. As discussed in the beginning of this chapter, this is one of the proposed problems (Problem 3) that needs to be handled by novaVM. Throughout this section, the main challenges of resource scalability are presented and then, current state of the art solutions are discussed.

Resource scalability is one of the key features in cloud computing; it allows host engines[5] to dynamically adjust the amount of allocated resources to meet changes in applications' workload demands [3]. 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 section, we focus on the later one.

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.

The "pay-as-you-use" model is specially interesting for a number of use cases. One clear example are applications that have diurnal patterns, i.e., most of the requests are issued and processed during the day, while users are active. For these applications, most resources used

[5] In the context of this work, a host engine refers to a virtualization engine which can be either a virtual machine hypervisor or a container engine.

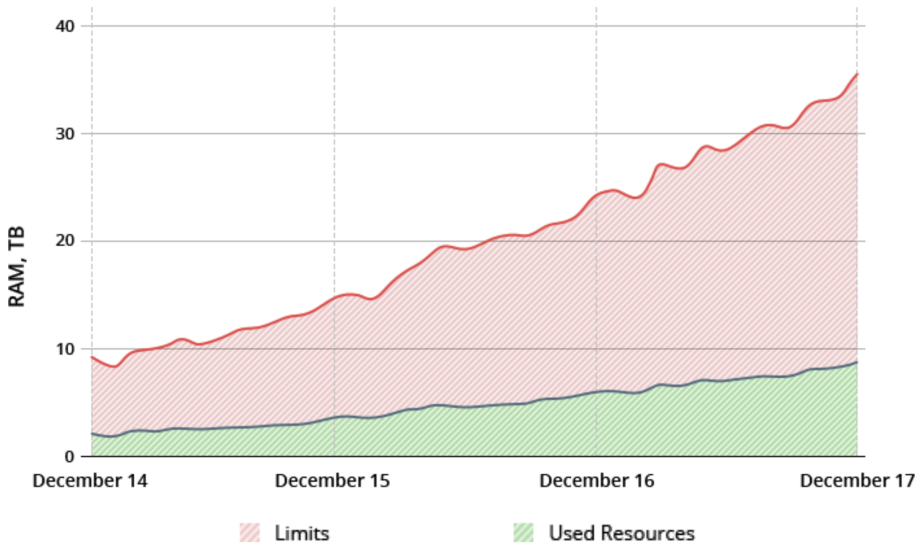

Figure 3.3: Jelastic Reserved vs Used Container Resources

during the day are not necessary during the night. Another similar example are applications which process data gathered during the day. These applications might use minimal resources during the day while, at night, more resources are required. Finally, any periodical task is a good candidate for this model, as resources might be only/mostly needed while the task is active. In sum, for all these types of applications, the ”pay-as-you-use” model enables not only cloud users to save significant amounts of resources (and therefore money), but also cloud providers to better use their machines (e.g., to support more applications in the same physical node).

Figure 3.3 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).

However, 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 and an application running inside an instance (JVM, container, or system-VM) must support vertical scalability.

This problem is even aggravated for applications that have different memory requirements throughout its execution [131]. The challenge is then to dynamically assign an 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.

Previous works 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) memory balancing in virtualized environments (i.e., managing memory assigned to instances), or ii) resizing the JVM heap. In the next sections, both perspectives are discussed and compared.

3.3.1 Memory Balancing in Virtualized Environments

Both virtual machines and containers support dynamic changes to the memory assigned to it. Determining the real memory requirements at runtime has been the focus of many previous works.

Waldspurger et al. [121] 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. [132] 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. [66] 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. [86] 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. [131] 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 sub-set of memory pages. Finally, Caballer et al. [26] 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.

3.3.2 Heap Resizing

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 [20, 122].

Although heap sizing is a well-studied problem, researchers are still looking for better approaches/trade-offs for this problem. Brecht et al. [20] propose a heuristic-based heap sizing mechanism for the Boehm collector [14]. 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 [14].

Yang et al. [125, 126] 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 [55] 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. [58] 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. [122] 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.

3.3.3 Resource Scalability Comparison

Table 3.3 summarizes the comparison of the previously described algorithms along three aspects: i) its type (if the algorithm is focused on improving memory balancing, heap resizing, or

Algorithm	Type	Vertical Scalability	Host Engine
Waldspurger et al. [121]	Memory Balancing	no	changed
Zhou et al. [132]	Memory Balancing	no	changed
Jones et al. [66]	Memory Balancing	no	changed
Lu et al. [86]	Memory Balancing	no	changed
Zhao et al. [131]	Memory Balancing	no	changed
Caballer et al. [26]	Memory Balancing	no	changed
Brecht et al. [20]	Heap Resizing	no	no changes
Grzegorcyzk et al. [55]	Heap Resizing	no	no changes
White et al. [122]	Heap Resizing	no	no changes
Hertz et al. [58]	Mixed	yes	changed
Yang et al. [125, 126]	Mixed	yes	changed

Table 3.3: Taxonomy of Resource Scalability Algorithms

even both); ii) if it supports memory vertical scalability by allowing the host engine to cooperate with the VM/container/JVM; and iii) if it requires host engine changes.

Analyzing previous approaches, it is possible to conclude that most solutions either improve on the host engine memory management system or improve on the instance (i.e., JVM) memory management system. This does not completely solve the problem of vertical scaling as there is a fundamental need to coordinate both the host engine with the instance in order for them to exchange memory as required by the applications. Only a few works include solutions that try to make the host engines cooperate with the heap resizing engines [57, 125, 126]. 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 inexistence of mechanisms inside the JVM that would allow the JVM and the host engine to exchange memory as required.

3.4 Summary

This chapter analyzed current approaches that try to solve each of the three proposed problems in Chapter 1. For each problem, a set of research works were described and compared with each other to identify problems and research opportunities.

From our VM migration algorithm analysis we can conclude that no current process migration algorithm is able to avoid unreachable data and therefore must migrate data that is no longer necessary. With regards to system-VM migration, some solutions support filtering unreachable data but still force the migration of state that does not belong to the target JVM that needs to be migrated (for example, the OS kernal and other processes).

For our Big Data memory management analysis it was possible to conclude that currently there is no GC approach that is able to reduce object copying (the main contributing factor for long application pauses) with no significant throughput overhead and developer effort.

Finally, in our resource scalability comparison we could observe that currently it is not possible to easily coordinate the host engine with the instance in order to allow the instance to scale vertically. Some existing solutions support vertical scalability but require changes to the host environment, something that is very had to achieve in real cloud provider scenarios.

In conclusion, for each of the proposed problems, after studying current related work, it is possible to conclude that current approaches are not ideal as they either do not completely solve the problem that they are targeting, or fail to comply with the proposed requirements (described in Chapter 1). This conclusion further motivates the following chapters, where we present algorithms that improve current research state.

Chapter 4

Architecture

In this chapter, novaVM's architecture is described in detail. First, its global architecture is presented and explained, discussing how algorithms integrate into the existing JVM. Then, from Section 4.2 to Section 4.6, each proposed algorithm is presented and discussed.

Globally, at a high level, novaVM comprehends three main ideas, each of which solving one of the problems mentioned in Chapter 1. First, improve horizontal scalability by providing a better migration and replication solution for JVMs (Problem 1, handled by ALMA in Section 4.2). Second, providing better application latency scalability by improving the GC and allowing the performance of applications to improve as more resources are granted (Problem 2, handled by NG2C, POLM2, and ROLP in Sections 4.3 to 4.5). Third, improve vertical memory scalability by improving resource management inside the JVM (Problem 3, handled in Section 4.6).

It is important to note that, as discussed in Chapter 1, both POLM2 and ROLP are profilers (with different granularities and performance trade-offs) that help novaVM achieving the requirements initially imposed, in particular, the requirement of not relying on programmer effort and knowledge (which is mandatory to use NG2C if no profiler is used).

4.1 Global Architecture

Each of the proposed ideas led to a separate algorithm which can be implemented as a sub-component and then added to the JVM. novaVM is the result of the integration of all these sub-components (that originated from the proposed ideas).

Figure 4.2 presents a very high level overview of novaVM. It inherits many functionalities from an already existing JVM implementation (OpenJDK) and thus, some components remain unchanged, while others were changed, and even new components were added. In Figure 4.2, blue components represent unchanged components; orange components are components that

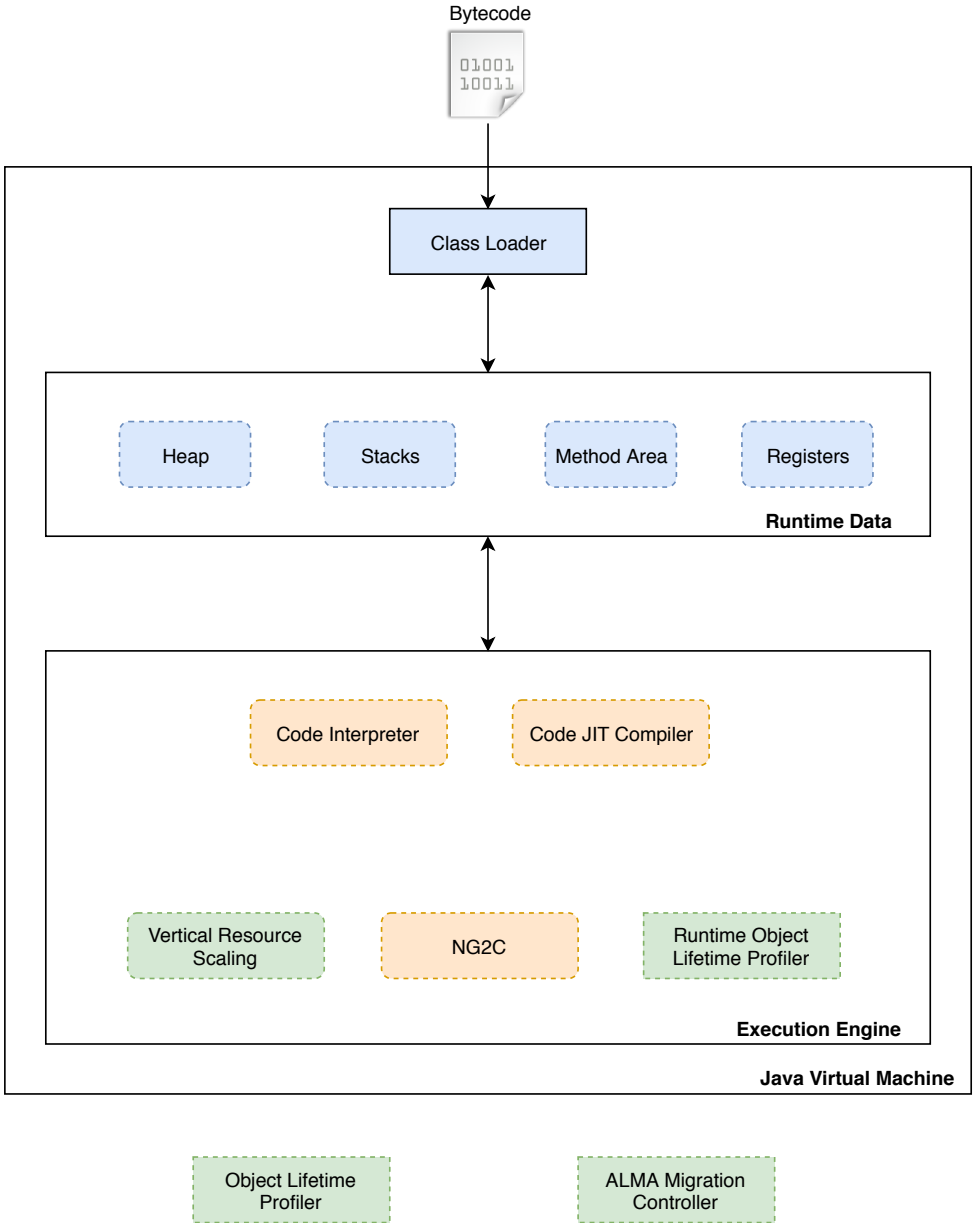

Figure 4.1: novaVM architecture

were changed, taking advantage of already existing code; green components represent new components.

Most JVMs, thus also including novaVM, can be decomposed in three main components: the class loader, the runtime data, and the execution engine. Most changes introduced by novaVM (in comparison to OpenJDK) rely in the execution engine component, which is responsible for coordinating the execution of applications (code compilation/interpretation and memory management, among others).

Looking at Figure 4.2, it is possible to identify which components accomplish which main goal/idea. First, the ALMA Migration Controller is the component responsible for taking advantage of the GC internal information to improve the migration or replication of the JVM (more

details in Section 4.2). The Vertical Resource Scaling component is the one responsible for taking better resource management decisions and therefore improve the vertical scalability of the JVM (more details in Section 4.6). The NG2C component represents the enhanced GC that allows an application to improve its performance as more resources are granted (more details in 4.3). Finally the Object Life Time Profiler and Runtime Object Life Time Profiler are, respectively, the offline and the online profilers that help users to take advantage of NG2C (see Sections 4.4 and 4.5 for more details). Note that NG2C can be used with no profiler. In such a scenario, in order to take advantage of NG2C, the programmer must manually annotate the code of the application and/or platform to give hints to the GC about the application objects' life time. In the remaining of this chapter, each algorithm is analyzed in detail.

4.2 ALMA: GC-assisted JVM Live Migration

This section presents ALMA, a JVM live migration algorithm that takes advantage of GC internal state to reduce the size of the snapshot to transmit between the source and destination sites. This algorithm, tackles Problem 1 (described in Chapter 1): the need to quickly recover from failed nodes or two spawn more nodes to accommodate new workload demands.

We start by giving a small introduction to ALMA's migration workflow and how it minimizes the amount of data to transmit during the migration using GC information. Then, we describe ALMA's architecture, how it reduces the snapshot size (by collecting parts of the heap) before a migration, the migration worklow, and finally, a set of optimizations.

ALMA's JVM live migration uses the following workflow (this workflow is described in further detail in Section 4.2.2): 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, takes an incremental snapshot of the JVM, and sends it to the destination site. The algorithm described in Section 4.2.2 works with any number of incremental snapshots.

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 per amount of time) that is superior to the network bandwidth; such regions will be collected to reduce their size.

ALMA is composed by two components, each one used on both source and destination nodes/sites (see Figure 4.2): Migration Controller, and JVMTI agent (JVM Tool Interface, described below). Both the destination and source sites are represented using dashed lines. Each process is represented with a gray background. JVMTI agents are represented by dotted lines.

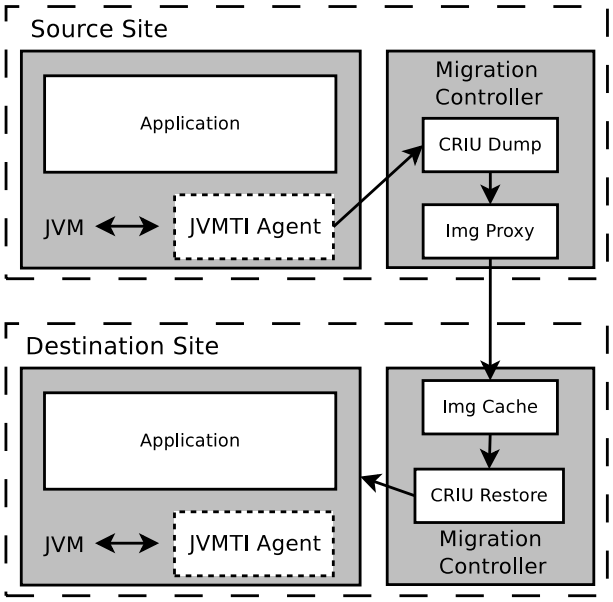

Figure 4.2: ALMA architecture.

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, which is described next); 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 incremental memory modifications 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 5.2.3.

The JVM was modified to contain a migration aware G1 GC policy. This policy is used, when a migration starts, to determine the segments of the heap 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.

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[1]. 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 4.2.2).

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

4.2.1 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 [91] rather than reference counting [32] 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 marking operations to collect more data nor introduce new GC data structures.

As already mentioned, G1 (discussed in Section 3.2.2) periodically marks the heap and produces several metrics per heap region (as discussed in Section 3.2.2, G1 divides the heap in equally sized blocks called regions), that result from the heap marking cycle, 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 spending the time to do so. We call the set of regions selected for collection *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) from the regions included in *CS* (see Eq. 4.1).

$$Data = \sum_{Heap} used(r) - \sum_{CS} dead(r) \tag{4.1}$$

Collecting a set of regions has a cost (time), which is defined in Eq. 4.2 as the sum of the cost of collecting each region in *CS*.

$$GCCost = \sum_{CS} cost(r) \tag{4.2}$$

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. 4.3).

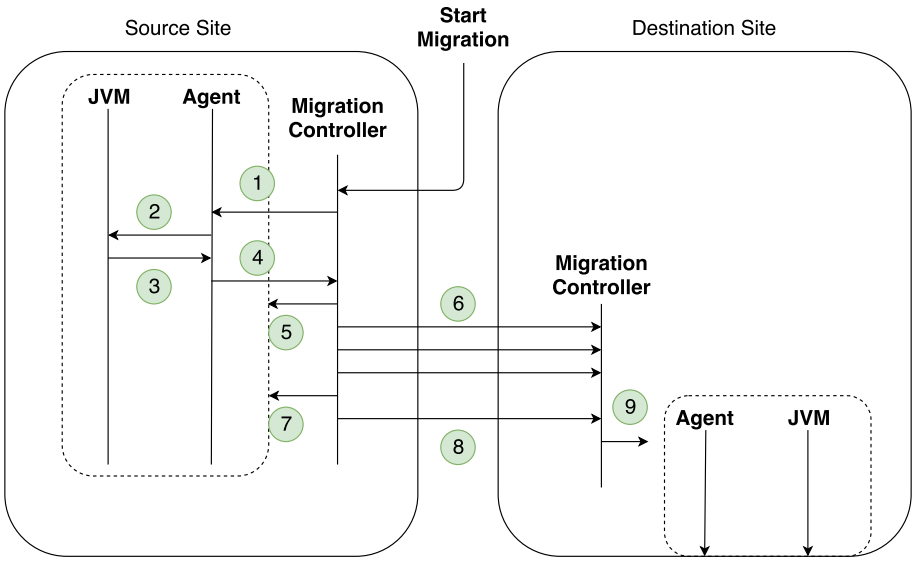

Figure 4.3: ALMA's Migration Workflow

$$MigrationCost = \frac{Data}{NetBandwidth} + GCCost \tag{4.3}$$

Taking into account that we want to minimize the migration cost by properly selecting regions for the *CS*, we must minimize Eq. 4.3. In other words, we need to maximize the amount of reclaimable space (i.e., minimize *Data*) and minimize the cost of collecting it (*GCCost*). Hence, we define the ratio *GCRate* (see Eq. 4.4) as the amount of data reclaimed per amount of time for a region *r*.

$$GCRate(r) = \frac{dead(r)}{cost(r)} \tag{4.4}$$

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. 4.5: all regions whose *GCRate* is greater than the *NetBandwidth* are selected for collection.

$$CS = \{\forall r : GCRate(r) > NetBandwidth\} \tag{4.5}$$

4.2.2 Migration Workflow

Having explained how the heap is analyzed and prepared for migration, we now describe ALMA's 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 4.3 represents this workflow. Note that, at start, the Migration Controller must be running both at the source and at the destination sites. Then, migration starts when the `Start Migration` event is received. The Migration Controller spawned at the source site is responsible for asking the JVM to prepare for a migration (step 1). This request is forwarded by the Agent to the JVM, that triggers a heap analysis, which results in the construction of the *CS*, which is then collected (step 2). The request is then answered (again using the Agent as intermediary) with a list of virtual memory ranges that contain no live data (steps 3 and 4). Note that these virtual memory ranges can be as large as a full heap region, but can also be smaller. This ensures that mostly live data is transmitted and 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 (step 5) while the JVM is still inside the last stop-the-world pause after collecting the *CS* (i.e., no mutator thread is running). In this step (step 5), 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 (step 6). This snapshot is incremental with regards to the previous one (except if this is the first snapshot).

Next snapshots take the same approach until the last snapshot (step 7) is taken; then, the Migration Controller at the source site notifies the Migration Controller at the destination site to resume the JVM (step 8). At the destination site, the Migration Controller simply receives application snapshots which are kept in memory, and waits for the resume JVM request. Upon reception, it rebuilds the JVM and the process resumes (step 9).

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 normally (at least for a large set of applications that we experimented with) having more than two snapshots does not reduce the application downtime. 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.

4.2.3 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; and ii) avoid collecting regions included in previous snapshots to avoid increasing the size of the differential snapshot.

Avoid Unnecessary Collections

Depending on the mutator memory allocation 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 user 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.

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.

4.3 NG2C: N-Generational Garbage Collector

As discussed in Section 3.2, most GC algorithms are based on the well established assumption that most objects die young. Therefore, to take advantage of this assumption, the heap is

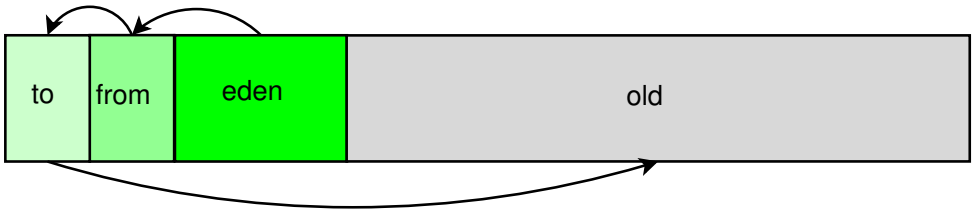

Figure 4.4: 2-Generational Heap Layout

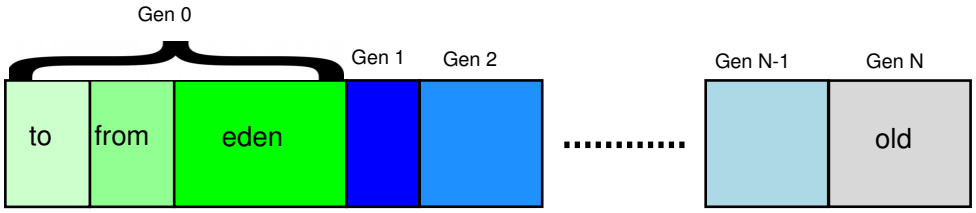

Figure 4.5: N-Generational Heap Layout

divided into two generations: young (where all objects are allocated in) and old (where objects which have lived for some time will be copied into).

As depicted in Figure 4.4, the young generation is composed by (green spaces): *Eden*, *To*, and *From* spaces. *Eden* is the space where objects are allocated in. The spaces *To* and *From* are used to hold survivor objects (objects that are still alive upon a minor collection).

When a minor collection (also called young GC) takes place, live objects are copied from *Eden* to one of the survivor spaces. Then, each object is copied between both survivor spaces (*To* and *From*) for a number of minor GCs (depending on the GC configuration). Finally, after being copied for a specific GC-dependent number of times, objects are finally copied to the old generation. From time to time, a full GC takes place; as the name suggests, a full GC collects all the generations, including the old, and lasts much longer than a minor GC.

This heap layout is appropriate for situations when most objects have a short life time. However, for middle and long-lived objects, the number of copies per object (between *To* and *From* spaces) can be significant because an object is copied whenever a minor/young collection takes place and the object is still reachable.

This copy process is bound to the hardware available memory bandwidth (which is a scarce resource in current commodity hardware). Another (bad) consequence of this copy process is heap fragmentation; this results from the fact that middle-lived objects with different life times are placed near each other (after being copied). With time, the heap gets fragmented as objects with shorter life times get unreachable.

More sophisticated GCs, NG2C in particular, use a N-generational heap layout (see Figure 4.5): objects are allocated in specific generations which contain only objects with similar life times. By grouping objects that will become unreachable approximately at the same time,

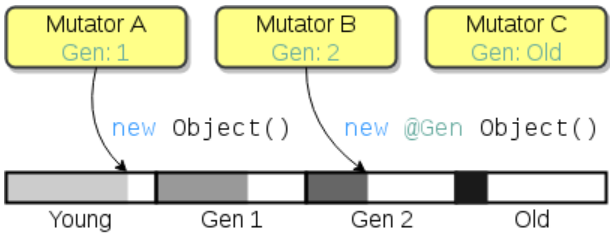

Figure 4.6: Allocation of Objects in Different Generations

NG2C avoids both object copying between generations and heap fragmentation. We now describe NG2C in detail.

4.3.1 Heap Layout

NG2C builds upon generational collectors's [5] idea but provides an arbitrary number of dynamic generations. The concept of dynamic generation is used instead of local/private allocation region because objects are grouped by estimated life time/age instead of being grouped by the allocating thread.

The heap is always created with two static generations: *Young* and *Old*. By default, all objects are allocated in the *Young* generation. Upon collection (more details in Section 4.3.4), live objects are copied to one of the survivor spaces or promoted to the *Old* generation. In other words, if no new dynamic generations are created, NG2C's heap layout is a 2-generational heap layout.

At runtime, any number of dynamic generations might be created (see Section 4.3.2 for more details). These dynamic generations are different from the static ones (*Young* and *Old*) in two ways: i) they can be created and destroyed at runtime, and ii) survivor objects are promoted directly into the *Old* generation.

In NG2C, objects can be pretenured into any dynamic generation and into the *Old* generation. With time, when objects become unreachable, the space previously allocated for a specific generation becomes available for other generations to use (more details in Section 4.3.4). In NG2C, except for the *Young*, the amount of heap space assigned to each generation is dynamic, increasing or decreasing as the amount of objects in that particular generation increases or decreases. This is possible since each generation is not implemented as a single large block of memory, but instead, as a list of memory regions (more details in Section 5.3).

Listing 4.1: NG2C API

```
1 // Methods added in class java.lang.System:
2 public static Generation newGeneration();
3 public static Generation getGeneration();
4 public static Generation setGeneration(Generation);
```

Listing 4.2: Job Processing Code Sample

```
1 public void runTask() {
2   Generation gen = System.newGeneration();
3   while (running) {
4     DataChunk data = new @Gen DataChunk();
5     loadData(data);
6     doComplexProcessing(data);
7   }
8 }
```

4.3.2 Pretenuring to Multiple Generations

NG2C is designed to profit from information regarding objects' life time profiles. This information is used to allocate objects with similar life times close to each other (i.e., in the same generation).

Since applications might have multiple threads/mutators managing objects with different life times (e.g., processing separate jobs), each thread must be able to allocate objects in different generations.

To efficiently support parallel allocation in multiple generations, we bind each application thread into a specific generation using the concept of current generation. The current generation indicates the generation where new objects, allocated with the `@Gen` annotation,[2] will be allocated into. In practice, when a thread is created, its current generation is the *Old* generation. If the thread decides to create a new dynamic generation, this will change the thread's current generation to the new one. It is also possible to get and set the thread current generation.

More specifically, the application code can use the following calls (see Listing 4.1):

- `newGeneration`, creates a new dynamic generation and sets the current generation of the executing thread to the newly created generation;
- `getGeneration` and `setGeneration`, gets and sets (respectively) the current generation of the executing thread. In addition, `setGeneration` also returns the previous generation.

To allocate an object in the current generation, the `new` instruction must be annotated with

[2] Starting from Java 8, the `new` instruction can be annotated. We use this new feature to place a special annotation that indicates that this object should go into the thread's current generation.

Listing 4.3: Data Buffer Code Sample

```
1 public class Buffer {
2    byte[][] buffer;
3    Generation gen;
4    public Buffer() {
5       gen = System.newGeneration();
6       buffer = new @Gen byte[N_ROWS][ROW_SIZE];
7     }
8 }
```

`@Gen`. All allocation sites with no `@Gen` will allocate objects into the *Young* generation (see Figure 4.6).

The code example in Listing 4.2 resembles a very simplified version of graph processing systems (e.g., GraphChi). It shows a method that runs several tasks in parallel threads. Each thread starts by calling `newGeneration`, to create a new dynamic generation. Then, while the task is not finished, all allocated objects using the `@Gen` annotation will be allocated in the new generation.

Listing 4.3 shows a code example that resembles a very simplified version of memory buffers in storage systems such as Cassandra; it shows how to use NG2C to allocate a large data structure (e.g., a buffer to consolidate database writes or intermediate data) while avoiding object copying. The constructor creates a new dynamic generation in which the buffer is allocated (using the `@Gen` annotation).

To sum up the two examples, generations are in fact being used as containers for objects with similar life times. Therefore, application threads can create and switch generations as many times as necessary. Note that, by default, i.e., without using the `@Gen` annotation or without creating or setting the current generation, all objects are allocated within the *Young* generation (in the *Eden* space). NG2C's extra functionality is mainly targeted to specific code structures that tend to allocate large portions of middle to long-lived objects. Such objects should be allocated directly in a separate generation to avoid not only useless and costly copies of these objects but also heap fragmentation.

4.3.3 Memory Allocation

NG2C allows each thread to allocate objects in any generation. This is fundamentally different from current HotSpot's allocation strategy which assumes that all newly allocated objects are placed in the *Young* generation. Hence, in order to support object allocation (pretenuring) into dynamic generations and into the *Old* generation, we extend the JVM's allocation algorithm.

In the JVM, object allocation is separated in two paths: i) fast allocation path, using a Thread Local Allocation Buffer (TLAB),[3] and ii) slow allocation path (very large object allocation or when the TLAB is full). Allocations through the `slow path` are handled in one of two ways: inside a TLAB (if there is enough free space), or directly in the current Allocation Region (AR)[4] (outside a TLAB). Note that for each generation, there is one AR.

The high level algorithm is depicted in Algorithms 3 and 4. For the sake of simplicity, and without loss of generality, we keep the algorithm description to the minimum, only keeping the most important steps.

A call to `Object Allocation` starts an object allocation. If the allocation is marked with `@Gen`, the allocation takes place in the current generation which is available from the executing thread state (otherwise the object is allocated in the *Young* generation). Objects are promptly allocated from the TLAB unless there is not enough space.

A call to `Alloc In Region` starts a large object allocation (or a TLAB if needed). Large object allocation (objects larger than a specific fraction of the TLAB size) goes directly to the current AR of the current generation (or to the *Young* generation if the allocation is not annotated). If the region has enough free space to satisfy the allocation, the object is allocated. Otherwise, a new region is requested from the available regions' list within the heap. If no memory is available for a new region, a GC is triggered followed by an allocation retry. If a GC is not able to free enough memory, an error is reported to the application.

The pseudocode for allocations in TLABs is not shown due to its high complexity and size requirements. Nevertheless, the code between lines 7 and 16 is already representative of how allocations inside a TLAB are conducted.

4.3.4 Memory Collection

In NG2C, three types of collections can take place. Figure 4.7 presents a graphical representation of these types of collections. In the figure, red space represents space that is no longer reachable, gray space represents space in dynamic generations that is still reachable, and black represents space in the old generation that is still reachable. In the following, we describe each type of collection in detail:

- Minor Collection: triggered when the *Young* generation has no space left for allocating new objects. Collects the *Young* generation. Objects that survived a number of collections (more details in Section 5.3) are promoted to the *Old* generation;

[3] A TLAB is a Thread Local Allocation Buffer, i.e., a private buffer where the thread can allocate memory without having to synchronize with other threads.

[4] An Allocation Region is used to satisfy allocation requests for large objects and also for allocating TLABs. Whenever an AR is full, a new one is selected form the list of available regions.

Algorithm 3 Memory Allocation - Object Allocation

```
1:  procedure OBJECT ALLOCATION
2:      size ← size of object to allocate
3:      klass ← class of object to allocate
4:      gen ← current thread generation
5:      isGen ← new instruction annotated with @Gen?
6:      if isGen then
7:          tlab ← TLAB used for generation gen
8:      else
9:          tlab ← TLAB used for Young
10:     if end(tlab) − top(tlab) >= size then
11:         object ← init(klass, top(tlab))
12:         bumpTop(tlab, size)
13:         return object
14: slow path:
15:     if size >= size(tlab)/8 then
16:         returnALLOC IN REGION(klass, size)
17:     else
18:         returnALLOC IN TLAB(klass, size)
```

Algorithm 4 Memory Allocation - Allocation in Region

```
1:  procedure ALLOC IN REGION(klass, size)
2:      gen ← current thread generation
3:      isGen ← new instruction annotated with @Gen?
4:      if isGen then
5:          region ← gen alloc region
6:      else
7:          region ← Young alloc region
8:      if end(region) − top(region) >= size then
9:          object ← init(klass, top(region))
10:         bumpTop(region, size)
11:         return object
12:     if isGen then
13:         region ← new gen alloc region
14:     else
15:         region ← new Young alloc region
16:     if region not null then
17:         object ← init(klass, top(region))
18:         bumpTop(region, size)
19:         return object
20:     else
21:         trigger GC and retry allocation
```

• Mixed Collection: triggered when the *Young* generation has no space left for allocating new objects and the total heap usage is above a configurable threshold. Collects the *Young* generation plus other memory regions from multiple generations whose amount of live data is low (more details in Section 5.3). Survivor objects from any of the collected memory regions are

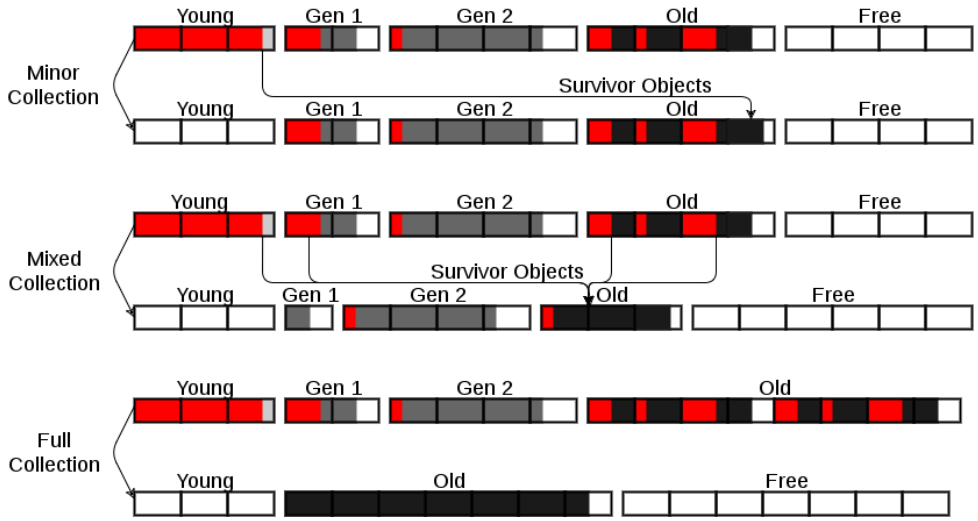

Figure 4.7: Types of collections (red represents unreachable data)

copied to the *Old* generation. Please note that, in a mixed collection, although all the regions belonging to the *Young* generation are collected, regions belonging to other generations are only collected if the percentage of live data is below a configurable threshold (the percentage of live data per region is gathered during a concurrent marking cycle, described next);

- Full Collection: triggered when the heap is nearly full. Collects the whole heap. In a full collection, all regions belonging to all generations are collected. All survivor objects are copied to the *Old* generation.

Note that when all regions that compose a dynamic generation are collected, the generation is discarded. If future allocations target a specific dynamic generation that was previously discarded, the target generation is re-created before the first allocation is actually performed.

Concurrent marking cycles are triggered when the heap usage exceeds a configurable threshold. During a marking cycle, the GC traverses the heap and marks live objects. As the name indicates, most of this process is done concurrently with the application. When the marking phase ends, the GC frees all regions containing only unreachable (i.e., unmarked) objects. For the regions that still contain reachable content, the GC saves some statistics (used for example in mixed collections) on how much memory can be reclaimed if a particular region is collected. As explained before, this information is used to decide which regions to collect in a mixed collection.

4.4 POLM2: Automatic Profiling for Object Life Time-aware Memory Management

In the previous section, NG2C was presented and analyzed. This new GC algorithm reduces object copying by grouping objects with similar life times in the same generation. However, it requires the programmer/developer to annotate the application code and to be able to estimate the life time of objects. Both are difficult and error-prone tasks. In this section, we present POLM2, a profile that will automatically create object life times estimates. To do so, POLM2 automatically instructs the GC to allocate objects with similar life times in the same generation. By optimizing object distribution in the heap (partitioning objects by estimated life time), POLM2 reduces GC effort to promote and compact objects in memory (both the two main causes for frequent and long application pauses).

POLM2 works in two separate phases (more details in Section 4.4.5): profiling, and production. First, during the profiling phase, the application is monitored in order to perceive its allocation profile. Secondly, in the production phase, the application runs in a production setting while having its memory management decisions taken accordingly to its allocation profile (output of the profiling phase).

NG2C is used to take advantage of the profiling information automatically extracted by POLM2. To take advantage of NG2C, POLM2 instruments the application bytecode (while it is being loaded into the JVM) to instruct NG2C on how to efficiently organize objects according to their estimated life time (thus reducing the GC effort, and consequently, application pause times number and duration). This is performed without any programmer intervention and with no source code modification. This process is explained in detail in the next sections.

4.4.1 Architecture

POLM2 is composed by several components which, combined, produce the application allocation profile (profiling phase), and change the application bytecode to give instructions to the collector regarding how objects should be placed in the heap (production phase). These two tasks are handled by four main components (see Figure 4.8):

Recorder - this component runs attached to the JVM where the application runs. It is responsible for two tasks: i) recording object allocations (i.e., the stack trace of the allocation plus the the object id, a unique identifier of the allocated object), and ii) informing the *Dumper* component on when it should create a new heap snapshot (more details in Section 4.4.2);

Dumper - upon request from the *Recorder*, creates a JVM memory snapshot. The memory

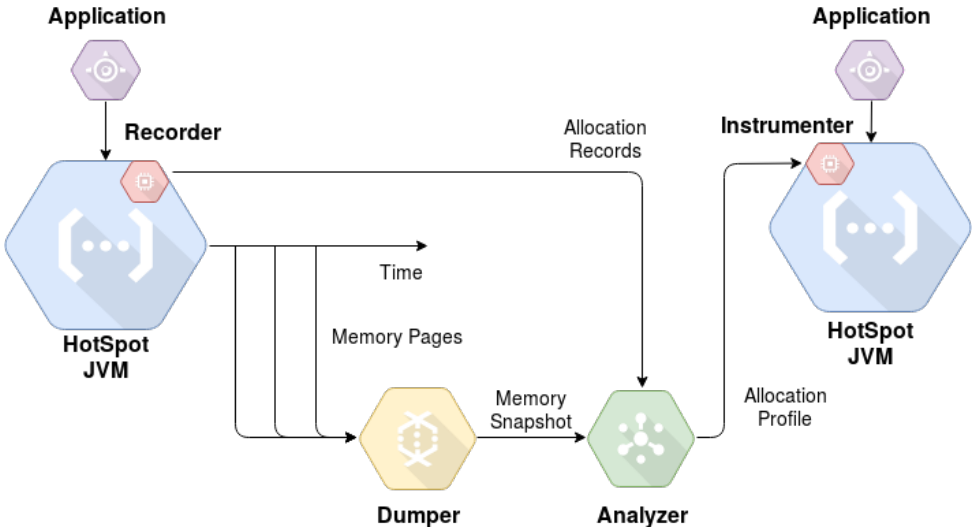

Figure 4.8: POLM2 Architecture and Workflow

snapshots are incremental (regarding the previous one) and do not include unreachable objects (more details in Section 4.4.2);

Analyzer - this component takes as input: i) the allocation records created by the *Recorder* (that states which objects were allocated and where) and, ii) the memory snapshots created by the *Dumper*. By analyzing a sequence of memory snapshots, the Analyzer is able to detect when objects start and stop being included in the snapshot, meaning that it is possible to perceive the life time of every application object. By combining this information with the allocation records, it is possible to estimate the life time distribution for each allocation site (more details in Section 4.4.3). This information constitutes the application allocation profile;

Instrumenter - taking as input the application allocation profile produced by the *Analyzer*, this component instruments (rewrites) the application bytecode while it is being loaded into the JVM. Based on the allocation profile, the *Instrumenter* instructs the collector on how to improve the distribution of objects in the heap by giving hints to NG2C on where to allocate objects (more details in Section 4.4.4);

4.4.2 Object Life Time Recording

To record object life times, two pieces of information are required. First, object allocations must be recorded, i.e., for each object allocation, both a unique object identifier (object id) and the stack trace of the corresponding allocation site must be included in the allocation record. Second, information regarding how long each allocated object lives is gathered through periodic memory snapshots, that include all live objects; thus, for each memory snapshot, it is possible to determine which objects are still reachable or not. By combining these two sources of

information (where objects are allocated, and how long they live), POLM2 is able to estimate, for each allocation site, the life time distribution of objects allocated through that allocation site. This information is then used to instruct NG2C.

Allocation records are created by the *Recorder*, a component that runs attached to the JVM. The *Recorder* has two main purposes. First, it instruments application bytecode (during application bytecode loading) to add calls to the *Recorder* logging methods on every object allocation. This ensures that whenever an application thread allocates an object, that thread will immediately (after allocating the object) call the *Recorder* code to log the allocation. Upon each allocation log call, the *Recorder* records the current stack trace plus the unique identifier of the object that is being allocated (more details on how this identifier is obtained in Section 5.4).

To avoid extreme memory and CPU overhead, the *Recorder* only keeps in memory a table with all the stack traces that have been used for allocations and continuously writes (to disk) the ids of the allocated objects (using a separate stream for each allocation site). Allocation stack traces are only flushed to disk at the end of the application execution (in the profiling phase). This ensures that POLM2 writes each allocation stack trace once to disk.

Apart from logging object allocations, the *Recorder* is also responsible for periodically requesting new memory snapshots (by calling the *Dumper* component). By default (this is configurable), the *Recorder* asks for a new memory snapshot at the end of every GC cycle. In other words, after each garbage collection (which collects unreachable objects), a new memory snapshot is created. In order to optimize this process of creating a memory snapshot (which can take a long time for large heaps and, since collections can occur very frequently), POLM2 offers two optimizations:

- the snapshot includes memory that contains only reachable data. To accomplish this optimization, the NG2C collector is modified to include an additional method call (accessible to the *Recorder*) that marks (more details in Section 5.4) all heap memory pages which contain no reachable objects (i.e., unused heap memory). Thus, before calling the *Dumper* for creating a new memory snapshot, the *Recorder* calls NG2C to mark all unused pages. Upon snapshot creation, the *Dumper* is able to detect which pages are marked or not, and simply avoids marked pages;
- only memory modified since the last snapshot is included in the next snapshot. Every time the *Dumper* creates a new memory snapshot, all memory pages that were part of the previous one, are marked clean (more details in Section 5.4). During application execution, changed memory pages are automatically marked dirty. Upon a new snapshot,

the *Dumper* is able to create an incremental snapshot that contains only the pages dirtied since the last snapshot. This results in much smaller snapshots (containing only modified memory pages) that are much faster to create.

Using these two optimizations, the time required to take a JVM memory snapshot is greatly reduced (evidence of this performance optimization is shown in Section 6.4). Thus, by reducing the time required to take a memory snapshot, POLM2 reduces the negative impact on application profiling (more details in Section 4.4.5).

4.4.3 Estimating Object Life Time Per Allocation Site

After profiling an application, the *Analyzer* can be started, taking as input: i) allocation records that include, per allocation site, the corresponding stack traces and allocated object ids (provided by the *Recorder*), and ii) memory snapshots (created by the *Dumper*). Using this information, it is possible to obtain an object life time distribution for each allocation site, i.e., the average number of objects that live for different amounts of time (measured in number of GC cycles). This object life time distribution enables the *Analyzer* to estimate the optimal generation to allocate objects per allocation site. This process is described next.

In order to determine the optimal generation for each allocation site, the *Analyzer* implements an algorithm with the following steps:

- process allocation stack traces (received from the *Recorder*) and, for each one, associate a sequence of buckets/sets (each one representing a generation);
- process allocated object ids (received from the *Recorder*) and insert them into the first bucket (generation zero) associated to the corresponding stack trace (where the object was allocated);
- process JVM memory snapshots (received from the *Dumper*), sort it by time of creation and, for each reachable object included in the snapshot, move the object id into the next bucket.

After these steps, it is possible to know, for each stack trace, how many objects survived up to N collections (where N represents the number of created memory snapshots). With this information, it is possible to obtain the number of collections that most objects allocated in a particular stack trace survive, which is an estimate for average number of collections that objects, allocated through a particular allocation site, survive.

However, one problem remains: it is possible to have two stack traces, with different estimated generations, sharing the same allocation site (remember that the allocation site must

Listing 4.4: Original Class1 Code

```
1  class Class1 {
2
3      public int[] methodD(int sz) {
4          return new int[sz];
5      }
6
7      public int[] methodC(boolean var) {
8          int[] arr = methodD(...);
9          if (var) {
10             int[] tmp = methodD(...);
11             ...
12         }
13         ...
14         return arr;
15     }
16
17     public void methodB() {
18         int[] arr;
19         if(...) {
20             ...
21             arr = methodC(true);
22             ...
23         }
24         else {
25             ...
26             arr = methodC(false);
27             ...
28         }
29         ...
30     }
31
32     public void methodA() {
33         ...
34         methodB();
35         ...
36     }
37 }
```

be annotated in order for NG2C to consider it for pretenuring). For example, if two different code locations use the same sequence of method calls to allocate objects with very different life times, both stack traces will share the final stack trace elements, thus creating a conflict. An example of such a scenario can be found in Listing 4.4, where method `methodD` is used by a sequence of methods that allocate objects with possibly very different life times.

To solve this problem, a stack trace tree (STTree) is built by the *Analyzer* to find a solution for such conflict. The STTree organizes stack traces as paths composed of a number of nodes. Each node is associated to a 4-tuple composed of: i) class name, ii) method name, iii) line number, and iv) target generation. Please note that POLM2 works at the bytecode level and therefore, the line number represents the corresponding bytecode index. The examples shown in this section are writen in Java for simplicity.

STTree nodes can either be intermediate (method call) or leaf (object allocation). Starting from any leaf, it is possible to reconstruct the allocation path (stack trace) of any allocation. By default, the target generation of all the intermediate nodes is zero (meaning that objects should be allocated in the youngest generation). Leaf nodes' target generation is obtained by using the estimated target generation that results from analyzing the objects included in the memory snapshots.

If one or more leaf nodes belonging to different sub-trees contain the same class name, method name, and line number, but different target generations, then it means that one or more conflicts exist. By design, these conflicting nodes belong to different sub-trees, meaning that it is possible to find at least one node in the allocation path that differs between each conflicting sub-trees. To solve existing conflicts, each of the conflicting nodes must push to its parent node its target generation. This operation must be repeated until parent nodes pointing to different code locations (combination of class name, method name, and line number) are found for each of the conflicting leaf nodes.

The resulting STTree for the code presented in Listing 4.4 is shown in Figure 4.9. Note that leaf nodes point to the same code location (`Class1`, `methodD`, line number 4) but contain different target generations (1, 2, and, 3, from left to right). To solve this conflict, each leaf node propagates its target generation to the parent node until a node pointing to a different code location is found.

The pseudocode of the algorithm used for detecting and solving conflicts is presented in Algorithm 5. The code is divided in two main procedures that detect and solve conflicts. Detection (*Detect Conflicts*) is done by searching for leaf nodes with identical 4-tuple values. Once a set of identical leafs has been found, the *Solve Conflicts* procedure is used to identify, for each

Algorithm 5 STTree Conflict Detection and Resolution

```
1:  procedure DETECT CONFLICTS(sttree)
2:      seen_4tuples ← empty
3:      conflicts ← empty
4:      for leaf in sttree do
5:          if leaf in seen_tuples then
6:              conflicts.add(leaf.4tuple, leaf)
7:              seen_4tuples.add(leaf)
8:      return conflicts

9:  procedure SOLVE CONFLICTS(sttree, conflicts)
10:     while conflict not empty do
11:         for node in conflict.nodes do
12:             conflicts.replace(node, node.parent)
13:         for node in conflict.nodes do
14:             if noConflic(node, conflict.nodes) then
15:                 conflicts.remove(node)
16:                 sttree.update(conflict, node)

17: procedure NOCONFLICT(tnode, nodes)
18:     for node in nodes do
19:         if tnode is not node then
20:             if same_code_location_different_generation(tnode, node) then
21:                 return false
22:     return true
```

leaf node, a parent node belonging to its allocation path which is unique. This parent node will be used to solve the conflict.

In Figure 4.9, each subtree associated with a different target generation is painted with a different color. For example, all leaf nodes that fall within the subtree painted in red (dotted line) will allocate objects in generation three. Also note that it is possible to override the target generation for a particular subtree, as it is illustrated in the subtree painted in yellow (dashed line), that allocates objects in generation one, although being part of the blue subtree (solid line, that allocates objects in the generation two).

4.4.4 Application Bytecode Instrumentation

Up until now, POLM2 is able (during the profiling phase) to profile and extract application allocation profiles. This section describes how to apply a given application allocation profile (during production phase) into the application bytecode at load time.

As mentioned before, POLM2 uses NG2C, the proposed GC that supports multi-generational pretenuring, meaning that it can pretenure (allocate) objects in multiple generations. NG2C provides a simple API that contains three methods: i) `newGeneration`, that creates a new

Listing 4.5: Class1 Code after Bytecode Instrumentation

```
1  class Class1 {
2
3      public int[] methodD(int sz) {
4          return new @Gen int[sz];                              // Added @Gen
5      }
6
7      public int[] methodC(boolean var) {
8          int[] arr = methodD(z);
9          if (var) {
10             Generation gen = setGeneration(Gen1);             // Line Added
11             int[] tmp = methodD(sz);
12             setGeneration(gen);                               // Line Added
13             ...
14         }
15         ...
16         return arr;
17     }
18
19     public void methodB(int sz) {
20         int[] arr;
21         if(...) {
22             Generation gen = setGeneneration(Gen2);           // Line Added
23             arr = methodC(true);
24             setGeneration(gen);                               // Line Added
25         }
26         else {
27             Generation gen = setGeneration(Gen3);             // Line Added
28             arr = methodC(false);
29             setGeneration(gen);                               // Line Added
30         }
31         ...
32     }
33
34     public void methodA() {
35         ...
36         methodB();
37         ...
38     }
39 }
```

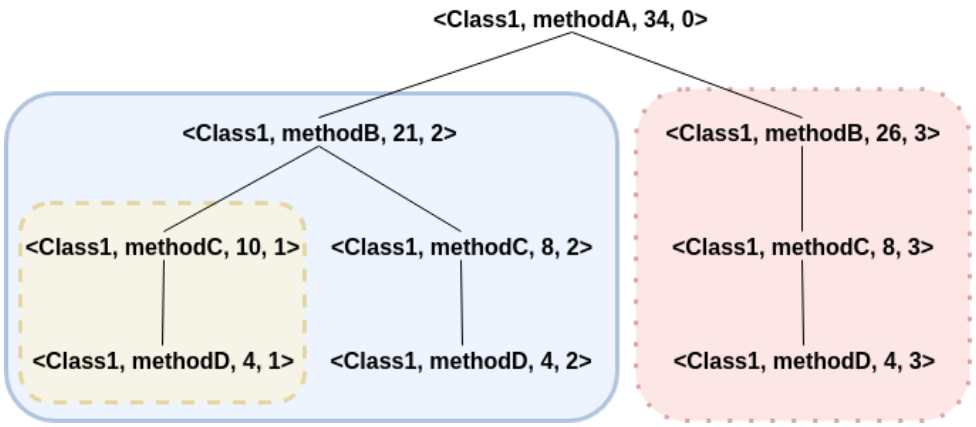

Figure 4.9: STTree for Class1 Source Code Allocations

generation; ii) `getGeneration`, that returns the actual target generation for allocations; and iii) `setGeneration`, that sets the current generation (and returns the previous one).

In addition to these methods, NG2C provides an annotation (`@Gen`) that is used to annotate allocation sites. Objects allocated through allocation sites annotated with `@Gen` are pretenured/allocated in the target generation (that was previously set with one of the methods of the provided API). Non-annotated object allocations are not pretenured, and are allocated in the youngest generation.

To automatically take advantage of an application allocation profile, POLM2 uses the *Instrumenter*, a component that intersects application bytecode loading and rewrites/instruments it to and insert calls (and annotations) to instruct NG2C. Following the previous example of Listing 4.4 and Figure 4.9, Listing 4.5 presents the Java code representation of the bytecode generated by the *Instrumenter*. Note that POLM2 does not change/require access to the Java source code; the listing is shown just for clarifying the description; it would be obtained if the bytecodes were disassembled. Note that, for clarity and simplicity, `try` and `finally` instructions are not present but would be necessary to ensure that `setGeneration` calls are not ignored due to unhandled exceptions.

Taking into consideration the application allocation profile, the *Instrumenter* added a `@Gen` annotation in line 4 that results in the pretenuring of an Integer array into the current generation. The current generation is controlled by calling `setGeneration` in lines 10, 12, 22, 24, 27, and 29. These calls are used whenever the execution steps into a subtree (from the STTree described in Figure 4.9) that contains a different color (target generation) regarding the current one.

The generations necessary to accommodate application objects (`Gen1`, `Gen2`, and `Gen3`) are automatically created (by calling the `newGeneration` NG2C API call) at launch time, and are available to use in any point of the application.

4.4.5 Profiling and Production Phases

As already mentioned, using POLM2 can be divided in two phases: i) profiling, and ii) production. The profiling phase is necessary to monitor the application in order to extract its allocation profile. During this phase, the *Recorder*, *Dumper*, and *Analyzer* components record allocation data and analyze it. The output of the profiling phase is an application allocation profile containing all the code locations that will be instrumented and how (annotate allocation site or set current generation).

The production phase represents the execution of the application in production environments. During this phase, only the *Instrumenter* component is required to instrument the bytecode according to the provided allocation profile. The instrumentation overhead is only present while the application is loading (during which the bytecode is being loaded and instrumented). After a short period, most of the code of the application is already loaded, and the performance overhead associated to the *Instrumenter* is negligible (see Section 6.4).

The separation between these two phases (profiling and production) is important for one main reason: a profiling phase generates an allocation profile for a particular combination of application and workload, meaning that whenever a particular workload is expected in the production phase, an already existing allocation profile can be used. This also means that it is possible to create multiple allocation profiles for the same application, one for each possible workload. Then, whenever the application is launched in the production phase, one allocation profile can be chosen according to the estimated workload (for example, depending on the client for which the application is running).

4.5 ROLP: Runtime Object Life Time Profiling for Big Data Memory Management

In the previous section, we described the approach taken in POLM2, in which the profiler runs off-line profiling of the application during the profiling phase, and then, in the production phase, applies the generated application allocation profile to instrument NG2C. This approach, however, has two potential problems. First, it requires the workload to be known in advance, something that might not be possible all the times. Second, it requires the workload to be stable, i.e., if the workload starts diverging from what was perceived at the profiling phase, the profiling decisions might no longer apply. To solve both those problems, in this section, we present ROLP.

ROLP is built to answer two simple questions: i) how long objects live, and ii) where (i.e., in

what code line) are objects allocated. In order to answer these questions, one must first define the notion of time and place. On the one hand, time is measured in GC cycles, i.e., the GC cycle is the unit of time (as it was in the previous section). Thus, the age of an object is the number of GCs that an object has survived. For example, upon allocation, all objects have an age of zero. As GC cycles occur, the age of surviving objects will increase by one unit at a time. On the other hand, ROLP defines the place of an allocation as an allocation context. An allocation context is a tuple of: i) an allocation site identifier, which identifies the line of code where the object is allocated, and ii) an execution stack state, which describes the state of the execution stack upon allocation (it will become more clear why this second item is necessary in the next sections).

4.5.1 Solution Overview

Following up the questions previously proposed, ROLP uses different approaches to handle each question. First, upon allocation, all objects are marked in their header with an allocation context that identifies both the allocation site (i.e., line of code) and the execution stack state. This piece of information, which is installed in an object's header, reveals where an object was allocated. Also note that if an object survives a GC cycle, and is copied to a another space (survivor or old), the corresponding allocation context will also be present in the object's header, in the new location.

With regards to knowing the age of objects (second question), ROLP tracks both the number of allocated objects, and survivor objects during GC cycles. This information (number of allocated and survivor objects) is kept in a global Object Life Time Distribution table (see Figure 4.10). This table maintains the number of objects with a specific age organized by allocation context.

In the next sections, we analyze in detail how ROLP is able to gather this information by both profiling application execution, and by tracking survivor objects during GC cycles. It is important to note that ROLP is designed to have ultra-low overhead, and be non-intrusive for the application's performance, as it is intended to be running inside the JVM during production workloads. Hence, all design decisions, including all the information gathered and all the instrumentation introduced in the application needs to be very simple and efficient.

4.5.2 Application Code Instrumentation

Following the design principle previously presented (i.e., all design decisions must be very simple and efficient), ROLP only profiles very frequently executed/hot application code. Thus,

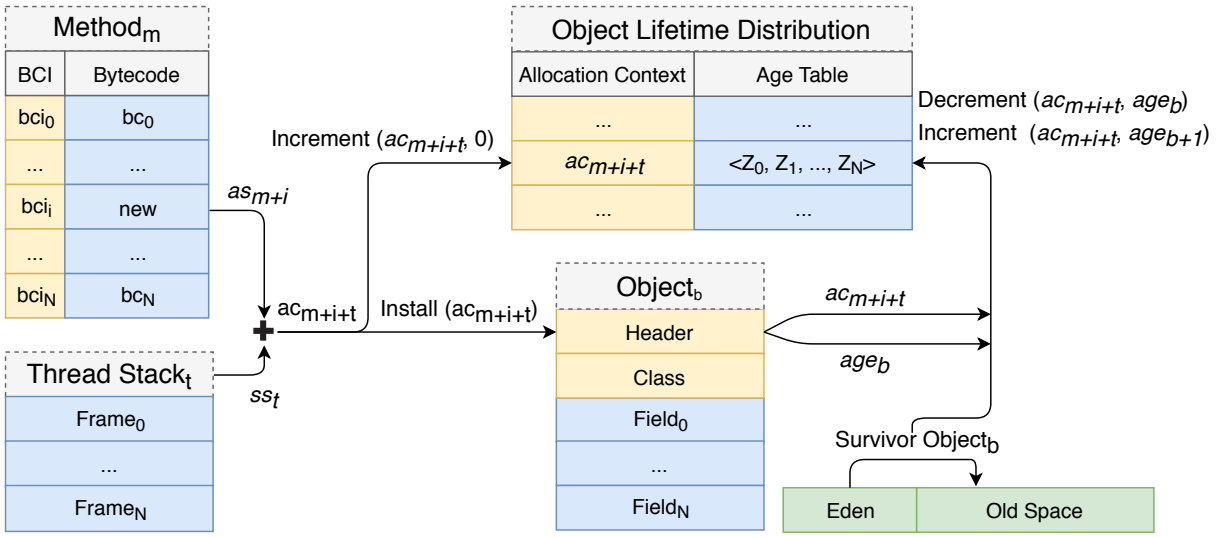

Figure 4.10: ROLP Profiling Object Allocation and GC Cycles

we take advantage of the JIT compilation engine in the HotSpot JVM to identify hot application code. In other words, we only insert profiling code in jitted code (i.e., native code compiled using the JIT compiler). There are two reasons behind this decision. First, installing profiling code has a cost (e.g., for creating unique identifiers for allocation sites) and thus, it makes sense to pay this cost only for application code that is executed very frequently (note that only a small fraction of the application code is usually hot). Second, since most of the execution time is spent running hot/jitted code, not profiling code that is not executed frequently (i.e., cold code), does not lead to a significant loss of information.

In short, the profiling code (added to the application code during JIT) is responsible for performing the following tasks: i) update the thread-local execution stack state whenever the execution stack is updated (whenever a new frame is pushed or removed from the stack); ii) increment the number of allocated objects (in the Object Life Time Distribution table) for the corresponding allocation context, upon object allocation; and iii) install the allocation context in the object header, upon object allocation. The next sections describe each one of these tasks in detail.

Updating the Number of Allocated Objects

The number of allocated objects per allocation context is maintained in the Object Life Time Distribution table (see Figure 4.10). As depicted in Figure 4.10, upon each object allocation, the allocation context (ac_{m+i+t}) is generated by combining both: i) the allocation site identifier (as_{m+i}), which identifies the specific code location where the allocation is taking place (this identifier is generated using the method and the bytecode index, BCI, where the `new` instruction

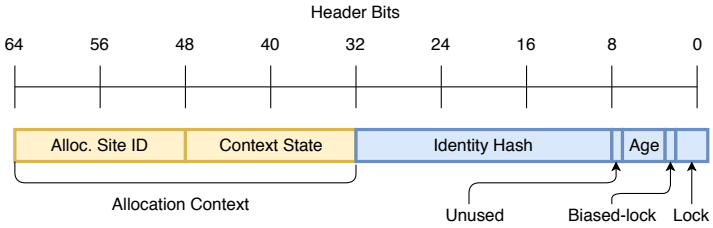

Figure 4.11: Object Header in HotSpot JVM using ROLP

occurs), and ii) the thread-local execution stack state (ss_t), which identifies the state of the execution of the application thread currently allocating the object. The resulting allocation context is installed in the header of the newly allocated object (this process is described next).

Marking Objects with the Allocation Context

Obviously, tracking an object's life time implies more than simply detecting when it was allocated. As a matter of fact, when a GC cycle runs, it is necessary to update the age of each survivor object (done by the collector) but also to update the number of objects which have survived another GC cycle in the Object Life Time Distribution table; for that purpose, ROLP must obtain the corresponding entry in the Object Life Time Distribution table. To do that, ROLP has to know each survivor object's allocation context.

Thus, ROLP associates each object with an allocation context by storing its corresponding allocation context in the object's header. Note that adding more information to application objects (for example, increasing the header size) is undesirable as it increases the memory footprint by adding extra bytes to every object. Therefore, ROLP reuses spare bits that already exist in an object header.

Figure 4.11 presents the 64-bit object header used for each object in the HotSpot JVM. The first three bits (right to left) are used by the JVM for locking purposes, followed by the age of the object (bits 3 to 6) which is also maintained by the JVM. Bit number 7 is unused, and bits 8 to 32 store the object identity hash, an object unique identifier. (See below details regarding the bit Biased-lock.)

As depicted in Figure 4.11, for each object, ROLP installs the corresponding allocation context in the upper 32 bits of the 64-bit header. These 32 bits are currently only used when an object is biased locked towards a specific thread,[5] and using them does not compromise the semantics of biased locks. Given that ROLP installs an allocation context upon an object allocation, if the object becomes biased locked, the profiling information will get overwritten. In

[5]Biased Locking is a locking technique available for the HotSpot JVM which allows locking an object towards a specific thread. It improves an object's locking speed for the presumably most frequent scenario as the object will only be locked by a single thread [38].

addition, biased locking is controlled by the JVM using a specific bit in an object header (bit number 3). Thus, if the object is biased locked (i.e., if bit number 3 is set) or if the allocation context is corrupted (i.e., it does not correspond to any entry in the Object Life Time Distribution table), the object is simply discarded for profiling purposes.

Using space dedicated to biased locks means that ROLP might lose some profiling information. However, through our experience and based on previous evaluation results, we argue that: i) the number of biased locked objects in Big Data applications is not significant; ii) data objects are usually not used as locks (and therefore are not biased locked); iii) not profiling non-data/control objects does not lead to a significant loss of important information since these control objects are usually small both in size and number.

In short, ROLP installs a 32 bit allocation context into each object's header. By doing this, ROLP is able to back trace the allocation context for any object. The allocation context may become unusable if the corresponding object becomes biased locked. In this situation, the object is not considered for updating the profiling information. However, this has no negative impact on ROLP effectiveness.

Allocation Context Tracking

The allocation context is a tuple of two elements: i) an allocation site identifier that identifies a specific line of code, and ii) an execution stack state. The execution state is necessary to distinguish two object allocations that, although using the same allocation site identifier (i.e., the same code location), use different call graphs to reach the allocation site (this problem was solved in POLM2 using STTrees). This is a very common scenario when object allocation and initialization is delegated to libraries or frameworks.

To track the execution stack state (or context), ROLP relies on the following. First, for allocation tracking purposes, it suffices that the execution state differentiates (as much as possible) two different call graphs. However, the details of the method calls that compose each call graph and their order (i.e., which method call was executed before the other) is not required to be contained in the execution state. Second, this state must be incrementally maintained as the application execution goes through the call graph and enters and leaves methods.

ROLP uses simple arithmetic operations (sum and subtraction) to incrementally maintain a 16 bit thread-local execution stack state. Thus, before each method call, the thread-local stack state is incremented with a unique method call identifier (hash value). The same value is subtracted when the execution exits the method.

Adding two arithmetic operations for each method call can clearly lead to throughput penal-

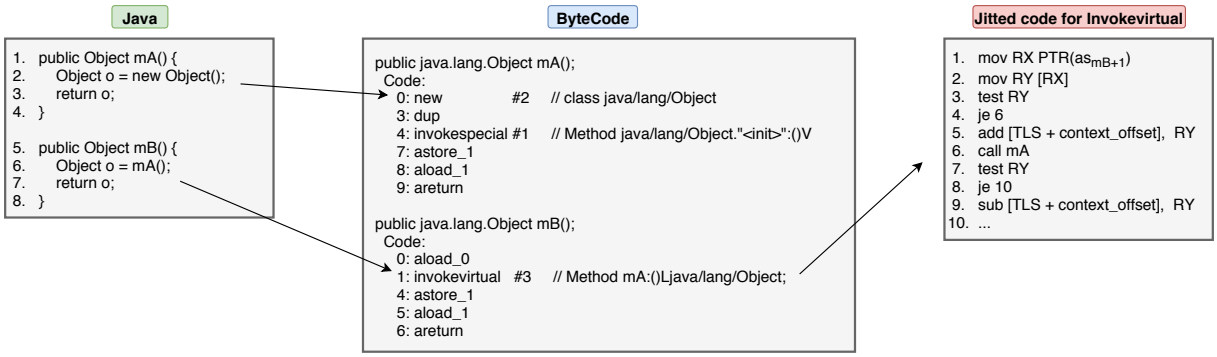

Figure 4.12: Code Sample: from Java to Bytecode to Assembly code

ties as method calls are very common in high level languages. In order to cope with this problem, ROLP is able to dynamically turn on and off the execution stack tracking for each method call. Hence, ROLP turns on this method call profiling code only for method calls that can differentiate call graphs leading to the same allocation site. This process is further discussed in Section 4.5.5.

Finally, it is also possible to have collisions in the execution stack state, i.e., if two or more different call graphs result in sequences of method calls, each contributing with a different value, lead to the same execution stack state. This problem is greatly reduced by two factors. First, we only profile hot code, thus greatly reducing the number of method calls that can contribute to a collision. Second, a collision would only be harmful if the allocation site is the same for the values that are colliding. Execution stack states that collide in different allocation sites are not a problem (i.e., they correspond to different lines in the Object Life Time Distribution table). Nevertheless, we demonstrate in Section 6.5.3 that collision are not a problem by showing that the number of allocation contexts with different life times is very low.

Code Profiling Example

In this section, we analyze an a snippet of code and see how ROLP installs the profiling code. Figure 4.12 presents a simple snippet of Java code (left), the result of its compilation to Bytecode using the Java compiler `javac` (center), and the Assembly code for the `invokevirtual` instruction produced by the OpenJDK HotSpot Opto JIT compiler (right). Both the Bytecode and the Assembly code presented in this figure are simplified for clarity reasons. We do not show the Assembly code that corresponds to the `new` instruction as it is more complex and would require too much space to illustrate with almost no benefit compared to analyzing the `invokevirtual` Assembly code.

The Java code snippet presents two methods, *mA* and *mB*. The first (*mA*) allocates an object which is returned to the caller (*mB*). The Bytecode corresponding to each method is

also presented. In this example both the `new` instruction and the `invokevirtual` instruction are profiled. The first (`new`) is profiled so that, for each object allocated in this allocation site, ROLP: i) increments the number of objects with age zero in the Object Life Time Distribution Table, and ii) installs the allocation context in the header of each object. The second (`invokevirtual`) is profiled to update the thread-local execution stack state. Note that ROLP also profiles all other variants of these instructions (that do not show up in this example) such as `newarray` or `invokespecial`.

For the remainder of this section, we analyze the the Assembly code generated for the instruction `invokevirtual` (right side of Figure 4.12). Lines 1 to 5 and 7 to 9 correspond to profiling instructions introduced by ROLP. These instructions are meant to increment (lines 1 to 5) and to decrement (lines 7 to 9) the thread local execution stack state by the specific amount that was calculated for this specific line of code (as_{mB+1}, note the byte code index 1 in method *mB* where the call is made). Note however that the increment or decrement Assembly instructions (`add` and `sub`) are executed on the condition that the value of as_{mB+1} is non-zero (note the `test` and `je` Assembly instructions in lines 3, 4, 7, and 8).

This conditional execution of the thread-local context update, enables ROLP to turn on and off the profiling of method calls. By doing so, ROLP avoids the execution of the `add` and `sub` instructions which can be costly as they may require loading and storing values to main memory (if the values are not cached). These instructions need to read and write to the current execution stack state which is stored `context_offset` bytes away from the Thread Local Storage (TLS, which is kept in a special register). Other than these two instructions (`add` and `sub`), only the `mov` instruction in line 2 requires memory access (which is much slower compared to operations performed using only registers or cached values). However, even for this instruction, which is necessary to load into memory the value that is added to the context state, we try to keep it in cache by storing it right next to the compiled code in memory. Thus, when the Assembly code of the method is loaded before it is executed, the value of as_{mB+1} will most likely be cached in the CPU, improving the performance of this instruction.

4.5.3 Updating Object Life Time Distribution Table

The information regarding the number of objects allocated per allocation context and age, is kept in the global Object Life Time Distribution table (presented in Figure 4.10). Besides being updated upon object allocation (to increment the number of objects with age zero), this table is also updated during GC cycles to update the number of objects that survived a GC cycle. In particular, let's assume an object allocated in the allocation context ac_{m+i+t} with age age_o that

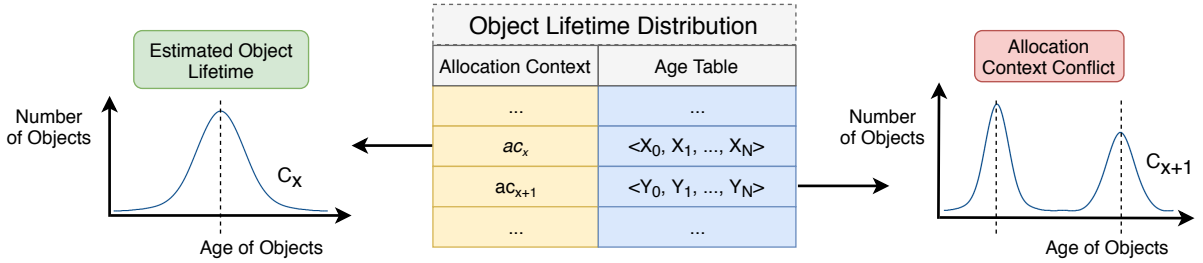

Figure 4.13: Extracting Curves from the Object Life Time Distribution Table

survives a GC cycle. The Object Life Time Distribution table will be updated to: i) decrement the number in the cell corresponding to row ac_{m+i+t} and column age_o (thus, one object less with age age_o); and ii) increment the number in the cell corresponding to row ac_{m+i+t} and column age_{o+1} (thus, one object more with age age_{o+1}).

This process is also depicted in Figure 4.10. In short, with ROLP, GC worker threads that are promoting survivor objects to the old space or to the survivor space will look into the objects header (see Figure 4.11) and extract the allocation context (upper 32 bits of the header) and the age of the object (bits 3 to 6). If the object is biased locked or if the allocation context is not present in the Object Life Time Distribution table, the object is not considered for profiling purposes. Otherwise, the worker thread will update the table and also increment the age of the object.

By the end of each GC cycle, the global table presented in Figure 4.10 contains the number of objects organized by allocation context and age. In order to ensure freshness, the Object Life Time Distribution table is periodically cleared. This operation is performed once every 16 GC cycles. This value is used because it is the maximum age of objects in HotSpot (considering that the age bits in the objects' header is only 4 bits long), after which, the age of the object does not increase more.

4.5.4 Inferring Object Life Times by Allocation Context

As already mentioned, ROLP maintains the number and age of objects per allocation context in the Object Life Time Distribution table. This table answers both proposed questions: i) how long do objects live, and ii) where (i.e., in what code line) are objects allocated. However, by combining the answer to the previous questions, one can answer a more interesting third question: what is the estimated life time of objects that will be allocated through a particular allocation context? Answering this question would allow ROLP to estimate the life time of an object before it is even allocated. As discussed before, such information would greatly benefit GC pretenuring algorithms as it allows objects with similar life times to be allocated close to

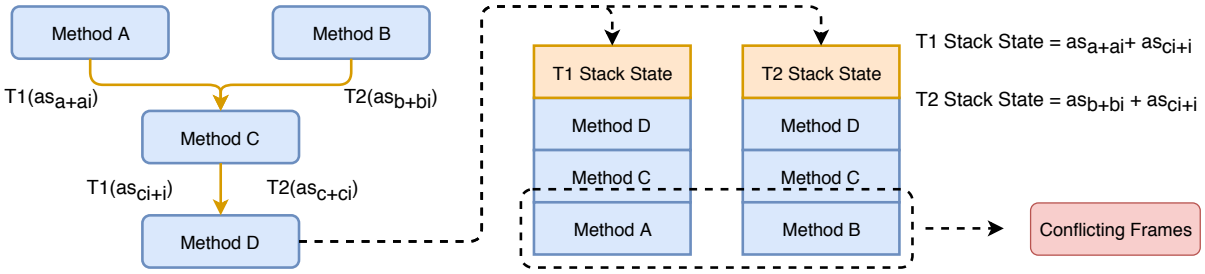

Figure 4.14: Thread Execution State on Allocation Context Conflicts

each other.

In order to infer the life time of objects allocated through a particular allocation context, e.g., ac_x, ROLP analyzes the number and age of the objects allocated through ac_x. To do so, a curve (C_x) plotting the number and age of (objects allocated through ac_x) is created. The resulting curve is most often very similar to a bell curve, whose maximum shows at which age most objects die. Hence, by determining the maximum of C_x, it is possible to infer with high confidence the estimated life time of objects allocated through ac_x.

It is possible, however, that a single curve (C_{x+1}, for example) shows not only one, but multiple bell-like shapes. Such a curve shows that the objects allocated through the allocation context ac_{x+1} may live for different amounts of time. In such a situation, we consider that we found a context conflict, which is possible if the same allocation site is being reached through multiple call graphs.

Figure 4.13 shows both the examples previously described. From the Object Life Time Distribution table, ROLP extracts curves for each allocation context. These bell-shaped curves are used to either extract the estimate life time of objects allocated through a particular allocation context (plot on the left of Figure 4.13) or to detect allocation context conflicts (plot on the right of Figure 4.13). In this specific example, the conflict is found by observing two bell shapes in the same curve (which indicates the existence of a conflict). The next section addresses this case, i.e. when an allocation context conflict occurs.

4.5.5 Dealing with Allocation Context Conflicts

Having seen so far how to detect allocation context conflicts, we now use this section to elaborate on how to solve these conflicts. As previously proposed, in order to answer the question of where an allocation takes place, it is not enough to simply identify the allocation site (i.e., code location) where the allocation occurs. This is so because, as already mentioned, multiple call graphs can lead to the same allocation site, and produce objects with very different life times (resulting in allocation context conflicts). Therefore, it is also necessary to track the execution

stack state, in order to differentiate two allocations that, although being triggered by the same allocation site, came from two different call graphs (i.e., their execution stack state is different).

Figure 4.14 illustrates such a situation. In this example, two threads, `Thread 1` and `Thread 2`, execute a sequence of methods calls until `Method D` is executed. Assuming there is an object allocation in `Method D`, it is possible that objects allocated by `Thread 1` have a different estimated life time compared to objects allocated by `Thread 2`, because they came through two different call graphs. In particular, before both threads converge in `Method C`, `Thread 1` came from `Method A` while `Thread 2` came from `Method B`. This can also be concluded by looking at the two thread stacks (Figure 4.14 shows that the stacks diverge on the bottom frame). Thus, in order to distinguish an object allocation that is being executed through the call graph of either thread, ROLP needs to track the context of the application.

However, tracking the context of an application (i.e, the execution stack state) is potentially harmful for the its performance as such tracking introduces a considerable amount of profiling effort to update the thread-local stack state. Therefore, a trade-off needs to be found. In one hand, not tracking the context means that ROLP would fail to determine where an allocation takes place (because the same allocation site will be reached through different call graphs); on the other hand, updating the context on every method call and return is very costly, and introduces undesired throughput overheads.

The sweet spot for this trade-off problem is achievable by identifying the minimum set of method calls that allows ROLP to distinguish different call graphs leading to the same allocation site. With such a minimum set of method calls, called *S*, it is sufficient to profile only the method calls in the set. In other words, ROLP only has to update the execution stack state when the methods calls in *S* are executed, thus avoiding conflicts with the minimum amount of throughput overhead.

Identifying such minimum set of methods (*S*) is not an easy task because of three factors/limitations. First, as mentioned in the beginning of this section, one of the requirements for ROLP is not to use any off-line information, such as a static analysis of the application source code (which can be impossible to obtain). Second, because of extreme use of polymorphism in languages such as Java, one cannot easily identify the callee of method calls at class load time or even at JIT compilation time. Third, adding profiling code to track caller to callee relations to help solving this problem is costly. In addition, as we demonstrate in Section 6.5, allocation conflicts are rare. In summary, the most important requirement for the technique that solves conflicts is the following: it must be very lightweight and non-intrusive w.r.t. application performance even if it means that it is suboptimal for solving the conflict within the minimum amount

of time.

Thus, the proposed algorithm to solve conflicts (and determine *S*) works as follows:

1. at JVM startup, no method call is profiled (i.e., a thread's local context is not updated when the thread enters or exits a method);

2. conflict checking is performed before the Object Life Time Distribution table is cleared (remember that this is done to ensure that information and statistics resulting from this table are fresh). Whenever a conflict is detected (multiple bell shapes in the same curve) *P* method calls are randomly selected to start tracking the thread-local context. *P* stands for an implementation specific number of method calls to profile at a time (we recommend that *P* should not be higher than 20 % of the total number of jitted method calls to avoid too much throughput overhead);

3. upon the next conflict checking, if the conflict was resolved, *S* must be contained in *P* method calls. In this case, ROLP can start to turn off method calls tracking until *S* is found. If the conflict was not solved, then a new set of *P* method calls must be selected (avoiding repeated method calls) and the process continues until all method calls are exhausted or until the conflict is resolved.

It is possible to have multiple sets of *P* methods being tracked at the same time, i.e., trying to solve multiple conflicts. Note, however, that *P* should be adjusted (reduced) as the number of parallel conflicts may increase so as to avoid high throughput overhead.

This algorithm presents two interesting properties. First, it is possible to dynamically control the number of method calls that are being tracked (or profiled) at a time while trying to resolve conflicts. Second, the algorithm converges on linear time to the number of jitted method calls divided by *P* and multiplied by the number of GC cycles between each conflict checking operation (16 GC cycles); this means that that it is possible to predict, on the worst-case, how long it will take to finish (we show this experiment in Section 6.5).

In short, as seen in the evaluation (Section 6.5), conflicts are rare. Therefore,in order to solve conflicts ROLP opted for using a very simple and low-overhead technique, as described above, opposed to using more performance intrusive but potentially faster techniques. In Section 5.5.1, we discuss how the information produced by ROLP is internally propagated to NG2C.

4.5.6 Updating Profiling Decisions

The lifetime of objects allocated through a particular allocation context can change over time if, for example, the application workload changes. To cope with these changes, novaVM needs

to continuously update its profiling decisions. Two types of situations are specially important for object lifetime profiling: i) if the lifetime of objects allocated through an allocation context increases, or ii) decreases.

On one hand, if the lifetime of objects allocated through allocation context ac_x increases, it means that objects allocated through ac_x are surviving more collections than before (and the Object Lifetime Distribution table will evidence that). This allows a pretenuring collector to take action and pretenure objects allocated by ac_x to an older space which is collected less often.

On the other hand, if the lifetime of objects allocated through ac_y decreases, the only visible effect is the increase in memory fragmentation (this information is updated by the collector at the end of each memory tracing cycle). When fragmentation is detected, novaVM identifies which allocation contexts are allocating objects in the fragmented memory regions and decrements their estimated object lifetime.

4.6 Dynamic Vertical Memory Scalability

Having described novaVM's proposed algorithms for Problem 1 and 2 (presented in Chapter 1), we now describe the proposed algorithms for tackling Problem 3: efficiently adapting JVM's resources to applications' needs.

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 were not designed to able to handle changes in the amount of resources given to them. We propose a re-design of the heap sizing strategy that allows 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 4.6.1). and ii) adapt GC heap sizing policies to better fit the application needs (see Section 4.6.2).

4.6.1 Letting the Memory Heap Grow

As discussed in Section 2.5, 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

Algorithm 6 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
```

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.

In order to allow the application to scale and to use more memory, the heap must keep growing. However, as discussed before (in Section 2.5.1), 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. In other words, the committed memory can grow until it reaches `CurrentMaxMemory`. By definition, `CurrentMaxMemory` is a subset of `MaxMemory` (reserved memory) and contains `CommittedMemory` (as depicted in Expression 4.6).

$$\texttt{CommittedMemory} \subseteq \texttt{CurrentMaxMemory} \subseteq \texttt{MaxMemory} \tag{4.6}$$

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 6.6.4, this overhead is negligible and the committed memory overhead is hardly noticeable.

Algorithm 6 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`.

In sum, 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`. Note that, as already said, this value can be also be changed programmatically.

4.6.2 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 (as discussed in Section 2.5.1) if no GC cycles are triggered (e.g., if an application does not need to allocate objects, or if an application is idle) it is not possible to scale down memory and thus, memory is kept in the JVM although it is not being used.

To solve the aforementioned 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 committed memory that is not being used), and ii) time since the last GC. The goal is to improve memory usage by uncommitting unused memory, but also not to disrupt the application execution by triggering very frequent collections.

Algorithm 7 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

Algorithm 7 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
```

least `MinTimeBetweenGCs` seconds ago (a configurable value).

Both `MaxOverCommittedMemory` and `MinTimeBetweenGCs` are configurable at runtime. By controlling these two variables, users can control how aggressive the JVM will reduce its heap size. It is important to note that, the more aggressive the policies are to scale down memory, 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 6.6.

4.6.3 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 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.

4.7 Summary

This chapter presented and analyzed a set of algorithms that improve the execution of Big Data applications on top of the JVM (novaVM). We started with ALMA, a live migration algorithm that

improves the way Big Data applications can recover from faults or migrate from one node to the other by reducing the amount of data included the snapshot by taking advantage of internal memory management state. By making use of this information, ALMA can ignore memory that is no longer usable by the application.

Secondly, we presented NG2C, a pretenuring GC algorithm that reduces long-tail latencies by avoiding object copies. This is possible by allowing the application to allocate objects with similar life times close to each other. By doing so, objects tend to die in clusters, reducing fragmentation and thus, avoiding object copies. However, NG2C requires source code modifications, which are not trivial in most scenarios. To solve this problem, POLM2 was presented as an off-line profiler that produces profiling information that can be used in production environments. ROLP is introduced as an online profiler that also avoids source code modifications. Compared to POLM2, ROLP is also able to cope with dynamic workloads, as it adapts its profiling decisions dynamically. As we will see through performance experiments, although ROLP is able to cope with these dynamic/unknown workloads, it introduces a slight performance overhead (whereas POLM2 does not).

Finally, our Vertical Memory Scalability algorithm was presented as a way to better adapt the memory usage of the JVM to the real application's needs. Our proposed algorithm periodically checks if the amount of over committed memory is high and if so, triggers a heap resizing operation. Additionally, it also allows the application to grow its memory beyond what was previously configured at launch time.

Chapter 5

Implementation

After presenting the design and architecture of novaVM, this chapter delves into a detailed implementation analysis of novaVM. The goal of this chapter is to provide to the reader the main implementation aspects that made possible implementing each of the algorithms described in the previous chapter in a production JVM.

We start by providing a global overview of novaVM, including a description of how each of the proposed sub-components interact with each other. Then, the implementation description is divided in multiple sections, each of which describing how a specific proposed algorithm (described in the previous chapter) is implemented as a sub-system of novaVM.

5.1 Global Implementation Overview

Our proposed system, novaVM, is an enhanced JVM for Big Data applications which tackles the problems and requirements presented in Chapter 1. Compared to its baseline implementation (OpenJDK 8 HotSpot JVM) novaVM provides a number of enhancements through multiple sub-systems. Each of the algorithms presented in the previous chapter were implemented in a separate sub-system of novaVM.

Our JVM uses the OpenJDK 8 HotSpot JVM as its baseline implementation, meaning that all sub-systems will either be a new component added to the JVM or will be implemented on top of an existing component.

ALMA (see Section 4.2) is implemented by modifying the garbage collector to be able to report object liveliness information to the ALMA Migration Controller. The ALMA Migration Controller was implemented on top of CRIU (see Section 3.1.1). NG2C was mostly implemented on top of the G1 collector (see Section 3.2.2). In addition, some modifications to the Code Interpreter, JIT Compiler, and Class Loader were also necessary to allocate objects di-

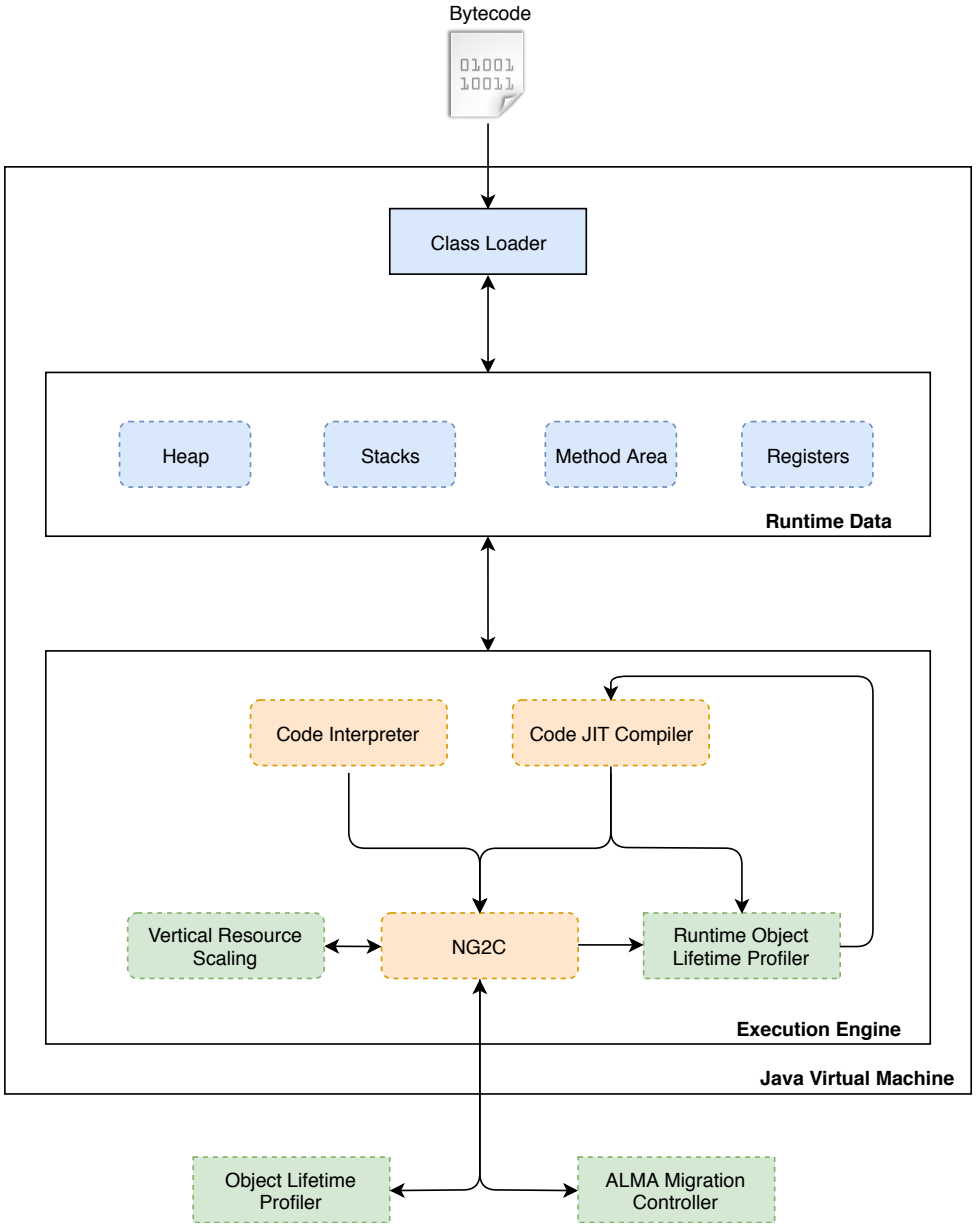

Figure 5.1: novaVM Component Interconnection overview

rectly in their target generation and to load `@Gen` annotations. POLM2 (see Section 4.4), on the other hand, was mostly implemented outside the JVM, using only JVMTI agents to either install profiling code, or to instrument code to take advantage of NG2C. Opposed to POLM2, ROLP (see Section 4.5) is completely implemented inside the JVM, introducing a new component to the JVM. Finally, our Vertical Resource Scaling sub-system (see Section 4.6) was implemented inside the JVM by modifying existing control structures inside the JVM.

Figure 5.1 presents a global implementation overview for novaVM, including the relations between the components introduced by novaVM. From the figure, it is possible to see that ALMA receives region liveliness information from the collector (NG2C). NG2C needs to handle the objects created by the Code Interpreter and by the JIT compiler and can also receive

instructions to resize the heap from the Vertical Resource Scaling module. This Vertical Resource Scaling module also takes as input the actions of the collector such as the size of the current heap, mutator allocation rate, etc. POLM2 (represented by the Object Life Time Profiler in Figure 5.1) takes as input the state of the heap, which is managed by NG2C. Finally, ROLP receives as input the state of the heap (mostly the survivor object counting) and the information inside the objects header (inserted by the JIT compiler) and produces as output instructions used by the JIT to direct object allocations to other generations.

The following sections describe in greater detail the implementation of each of the subsystems of novaVM. For each section, it is given special focus to the implementation decisions and performance optimizations that made possible the implementation of each proposed algorithm in a production JVM with very good performance.

5.2 ALMA's Implementation

In this section we describe ALMA's implementation details. First, we discuss an implementation overview for ALMA. Then, we present how our migration aware GC is implemented, and discuss the internal details of the Migration Controller. The Migration Controller is implemented on top of CRIU. Most of the code developed on top of CRIU has been integrated into the project's main repository[1] and is now available through CRIU releases.

5.2.1 Implementation Overview

Figure 5.2 presents an overview of ALMA's implementation components. Green components represent components that were introduced or modified by ALMA while white components represent components of the JVM that were not unmodified by ALMA. In particular, in order to implement ALMA, we had to i) introduce changes to the Runtime Core (component where most of the runtime logic resides), and ii) add two new components to the JVM, the Migration Aware Policy, and the Migration Agent. The first (Runtime Core) was modified to be able to interact with the Migration Agent (i.e., to receive and answer to migration aware GC calls). This Migration Agent is required allow the communication between the Migration Controller (described next) and the Runtime Core. Finally, the Migration Aware Policy (described in Section 5.2.2) is used to select the heap regions to collect before a migration.

In addition to the components added to the JVM, ALMA also provides a Migration Controller (described in Section 5.2.3) which handles the creation and transference of the JVM snapshots

[1] github.com/checkpoint-restore/criu

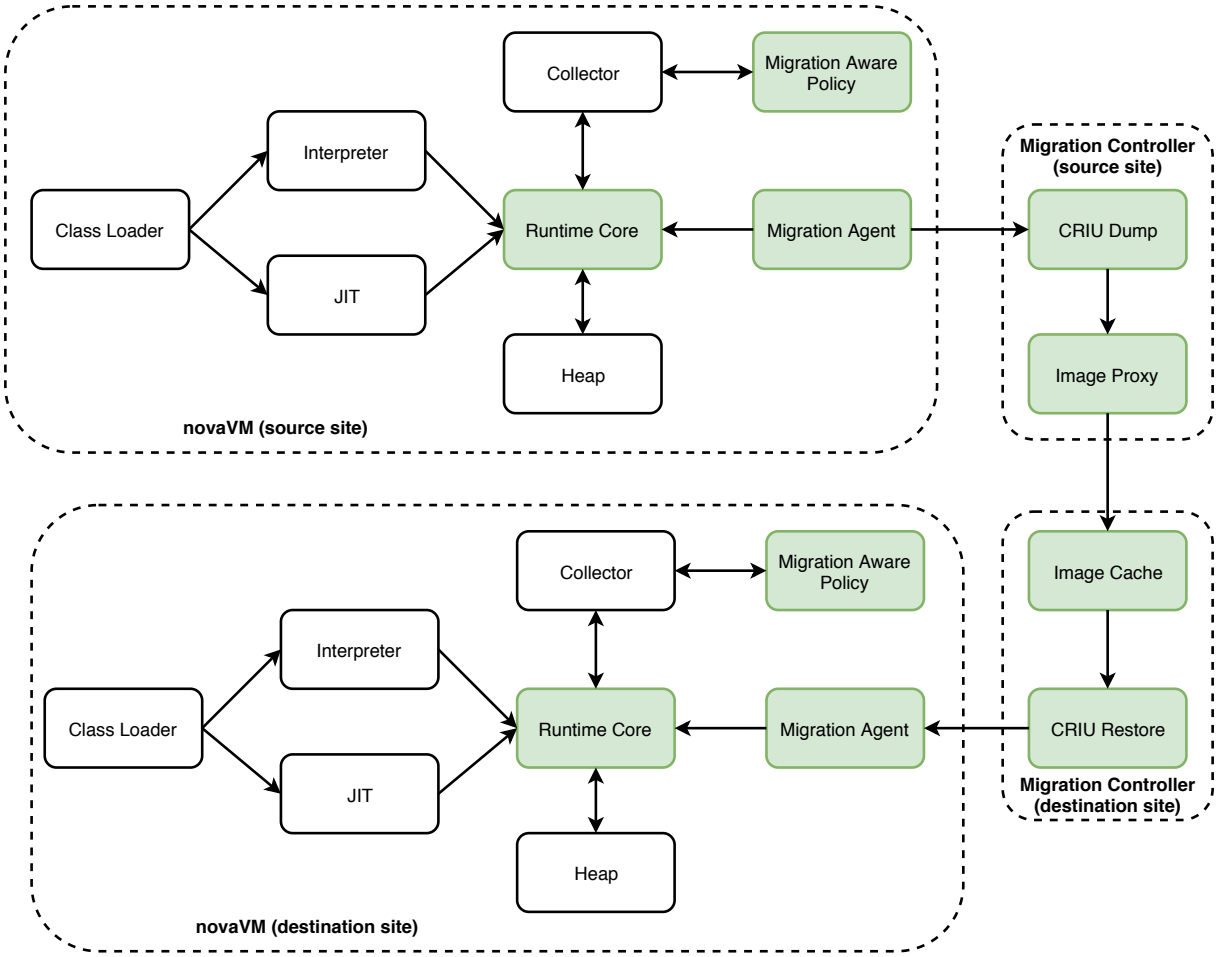

Figure 5.2: ALMA Implementation Components

to the destination site. In the next sections, we provide further details on how the Migration Aware Policy and Migration Controller are implemented.

5.2.2 Migration Aware GC

ALMA is implemented in novaVM (which is based on the OpenJDK HotSpot JVM 8) and integrates with NG2C (based on the G1 collector). ALMA adds a migration aware policy (presented in Section 4.2.1) which takes advantage of the already existing collector 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 collector uses, internally, data regarding each heap region (e.g., number of free bytes, used bytes, references from objects 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 collector is able to tell how many live bytes reside inside a particular region. As time goes by and regions get collected, the collector 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 the collector to compute the optimal set of memory pages to include in the snapshot. This heap analysis is ruled by the equations described in Section 4.2.1. In other words, only regions in which garbage can be collected faster than transmitted through the network are collected.

By the time a migration starts, some regions do not have yet neither an estimate for the number of bytes that can be freed nor the time it will take. Such a lack of information can happen because the mutator is not paused during the marking cycle (thus, it continues to allocate memory). However, this is not a problem since newly allocated regions (i.e., young regions) will most likely have the highest *GCRate*, according to the motivation behind generational garbage collectors (which predicts that most objects will die young [82]). Thus, when such a region is found (with no GC estimates), it is simply added to the *CS*. It is interesting to note that, for applications with very fast memory allocation ratios, the proposed migration aware policy very much like a simple young GC which only collects young regions.

5.2.3 Migration Controller

The Migration Controller component used in ALMA is implemented using CRIU (as presented in Section 3.1.1, CRIU is a checkpoint restore tool for Linux). We modified CRIU to: i) support live migration through the network (the 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's 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 (see Figure 5.2, where both these com-

ponents are illustrated). 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 pro-actively 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.

In addition, the Image Proxy is specially important because it keeps in memory all snapshots done so far in order to ensure that future snapshots will be incremental. If there was no Image Proxy at the source site, ALMA would have to fetch previous snapshots from the Image Cache (remote site) or store them in disk (which would be much slower).

5.3 NG2C's Implementation

NG2C is implemented on top of the Garbage First (G1) GC [37]. G1 is the most recent and advanced GC algorithm available for the OpenJDK HotSpot JVM 8. In addition, G1 is the new default GC in the HotSpot JVM. NG2C builds upon G1, by adding approximately 2000 lines of code. NG2C is integrated with G1 in the way that applications that do not use the `@Gen` annotation will run using the G1 collector (i.e., the code introduced by NG2C is never activated). This has the great benefit that all the effort put into developing G1 is leveraged by NG2C. For the rest of this section, we describe how we modified G1 for supporting pretenuring into multiple generations.

By using G1's as our code base, we inherit many techniques that are already well implemented and tested. In other words, we are using all the GC techniques already implemented in G1 (such as heap region management, remembered sets management, safepoints, write barriers, and concurrent marking) to support NG2C's implementation.

5.3.1 Implementation Overview

As presented in Section 3.2.2, G1 uses a heap divided in equally sized memory regions. It contains two generations, the *Young* and the *Old*. The first is divided into three spaces [116] (*Eden*, and two survivor spaces, *To*, and *From*). NG2C maintains both these generations with the exact same structure and semantics.

Additional dynamic generations are created by allocating regions from the free memory regions list (also available in G1). The existing code in G1 looks at NG2C's dynamic generations

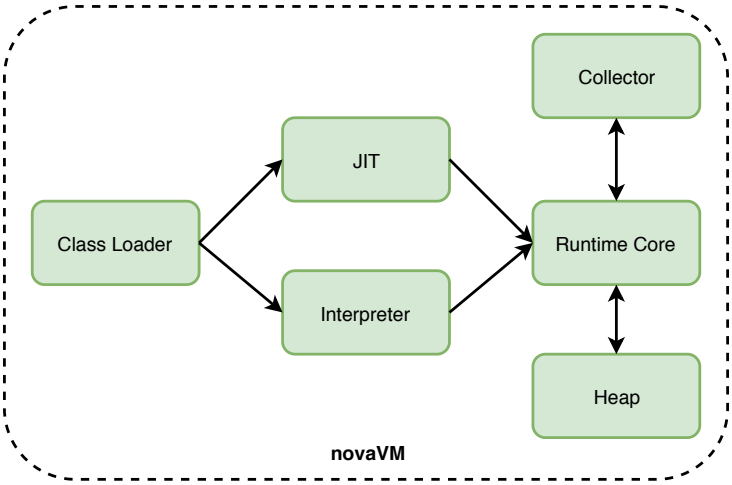

Figure 5.3: NG2C Implementation Components

as part of G1's *Old* generation. This means that we reuse G1's write barrier and remembered set for inter-generational pointers. This ensures both the correctness of the collection process in NG2C and also allows NG2C to take advantage of highly optimized code, developed initially for G1.

NG2C's inherits G1's collector algorithms with only minor changes (again, this ensures that NG2C is taking advantage of highly optimized and tested code). Minor, mixed, and full collections work in the exact same way in both NG2C and G1. The only modification is that, in NG2C, the collector can promote objects from dynamic generations into the *Old* generation, while in G1, the collector either only promotes from *Young* to *Old* or compacts regions belonging to the *Old* generation.

Most of the code introduced by NG2C, lies in the object allocation path that, however, leads to code changes through the whole runtime stack. As can be seen in Figure 5.3, NG2C implementation touches all main components of the JVM (green represent changed components): i) the Class Loader component was changed to process `Gen` annotations; ii) the JIT and Interpreter were changed to be able to pretenure objects; iii) the Runtime Core was changed to allow allocations of objects in multiple generations; iv) the Heap was changed to support multiple generations; v) and the Collector was changed to be able to collect different generations. In the next sections we describe in detail the most important aspects of our implementation. In particular we describe how the new allocation algorithm works and how the bytecode interpreter and JIT compiler are adapted to work with it. We continue in the next sections with an analysis of several important implementation aspects of NG2C.

5.3.2 Parallel Memory Allocation

Contention in memory allocation is a well-known problem [67, 47]; memory allocation must be synchronized between threads so that each memory block is used by a single thread. In G1 this is achieved by using Thread Local Allocation Buffers (TLABs) and Allocation Regions (ARs). Therefore, whenever a thread needs to allocate some memory, it allocates directly from its TLAB. If the TLAB is full, the thread must allocate memory from the current AR. This allocation, however, will only occur after the thread acquires a lock on that AR. If the AR does not have enough available space, a new AR is allocated directly from the list of free regions (this step requires even further locking to ensure that no other thread is allocating another region).

In NG2C, we extend the use of both TLABs and ARs to multiple generations (the complete algorithm is presented in Section 4.3.3). Since each thread can now allocate memory in multiple generations, multiple TLABs are necessary to avoid costly memory allocations. The TLAB to use for each allocation is decided at runtime, based on the use of `@Gen` annotations (see Section 5.3.4 for more details). Additionally to TLABs, NG2C also extends the use of ARs to multiple generations. Therefore, whenever a TLAB used for a particular generation is full, an allocation request is issued directly to the AR of the specific generation.

By using multiple TLABs and ARs (one for each generation), allocations are more efficient as fewer synchronization barriers exist compared to not using (TLABs and ARs). This, however, introduces a problem: as any thread can allocate memory in any generation, each thread must have a TLAB in each generation (even if that thread never allocates memory in that particular generation). As the number of generations grow, more and more memory would be wasted for allocating TLABs that were never actually used.

To solve the aforementioned problem, NG2C never actually allocates any memory for TLABs when creating a new generation. Memory for each TLAB is effectively allocated only upon the first allocation request. This way, threads will have TLABs (with allocated memory) only for the generations that are being used (and not for all the existing generations).

5.3.3 `@Gen` Annotations

We now move into the decision of why using `@Gen` annotations to select in which generation to allocation an object. For allocating memory in generations other than the *Young* generation, we considered several options: i) simply calling the JVM to switch the generation to use for allocation; ii) add a new `new` instruction with an extra argument (target generation); and iii) annotate the `new` instruction.

The first option was immediately ruled out because it is very difficult to control which objects

go into non-young generations; e.g., naïve `String` manipulation can easily result in many allocations that would potentially go into a non-young generation. The second option (creating a new allocation instruction) would force us to extend the Java language, and the compiler. Thus, we naturally opted for the last option.

A clear advantage of using annotations is its simplicity; however, it has one disadvantage: we must call the JVM whenever we need to change the current target generation. However, in practice and according to our experience, this almost never imposes a relevant overhead because: i) a thread handling a particular task will most probably only need one generation (worker threads tend to use one generation at a time), and ii) large object allocation and copying is much more expensive than calling the JVM to change the target generation (therefore it pays off to allocate a large object in the correct generation). In both cases, the cost of calling the JVM is absorbed and the overhead becomes negligible (see Section 6.3 where we show that NG2C does not decrease the application throughput). Also note that getting and setting the current generation does not require any locking as it only changes a field in the current thread's internal data structure.

5.3.4 Code Interpreter and JIT

The OpenJDK HotSpot uses a combination of code interpretation and Just-In-Time (JIT) compilation to achieve close to native performance. Therefore, whenever a method is executed for the first time, it is interpreted. If the same method is executed for a specific number of times (implementation specific), it is then JIT compiled. This way, the JVM compiles (a costly operation) only the methods where there is benefit (since executing compiled code is much faster than interpreting it).

In order to comply with such techniques in NG2C, we modify both the interpreter and the JIT compiler to add the notion of generations. To be more precise, we had to detect if the allocation is annotated with `@Gen` and, if so, which generation is being targeted (choose the correct TLAB).

Selecting the correct TLAB to allocate is done as follows. For each thread, NG2C keeps a pointer to the current generation TLAB. This pointer is only updated when the thread calls `newGeneration` or `setGeneration`. Then, if the current allocation site is annotated with `@Gen`, the current generation TLAB is used.

Detecting if the current allocation is annotated with `@Gen` is done differently before (interpretative mode) and after JIT compilation. Before JIT, NG2C uses a map of bytecode index to annotation, that is stored along the method metadata (this map is prepared during class loading). Using this map, it is possible to know in constant time if a particular bytecode index is

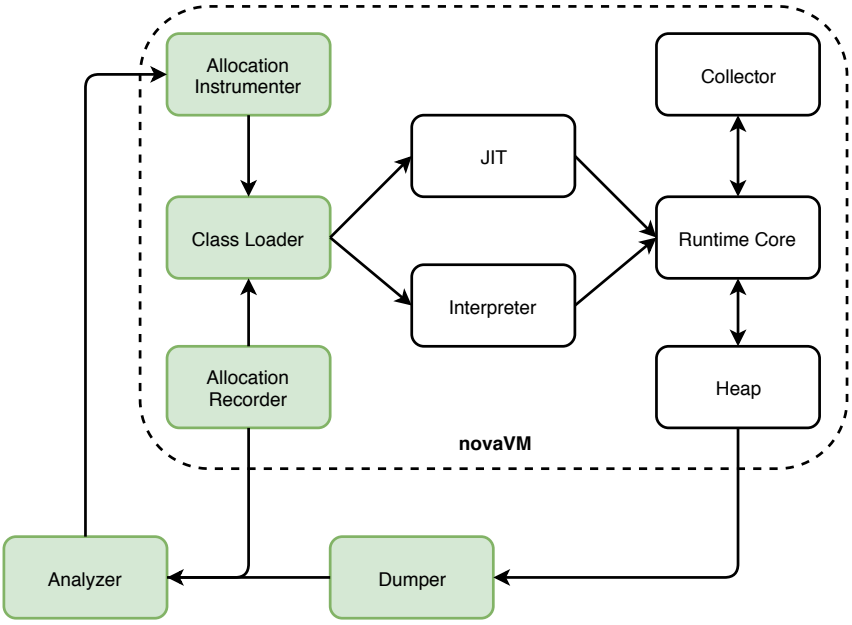

Figure 5.4: POLM2 Implementation Components

annotated with `@Gen` or not. Upon JIT compilation, the decision of whether to go for the *Young* generation or not is hardcoded into the compiled code. This frees the compiled code (after JIT) from accessing the annotation map.

5.4 POLM2's Implementation

POLM2 is composed of a collection of components that, as described in Section 4.4, work towards one common goal: automate the discovery of application allocation patterns that could be used to instrument the code to take advantage of NG2C. This section starts by providing an overview of the implementation followed by a description of the implementation techniques that are relevant for implementing POLM2.

5.4.1 Implementation Overview

As presented in Section 4.4, POLM2 is comprised by four main components: i) the Allocation Recorder (tracks object allocations during the profiling phase); ii) the Allocation Instrumenter (instruments allocations during the production phase to take advantage of profiling information); iii) the Dumper (snapshots the JVM for further offline object life time analysis); and iii) the Analyzer (produces profiling information based on the allocation records and dumps).

Figure 5.4 present the implementation components that were introduced or changed to implement POLM2 (components painted in green). In addition to the four main components just refereed, the Class Loader was also changed (as it was for NG2C) to be able to process

`@Gen` annotations. We could not use the same Class Loader modifications used in NG2C because automatically inserted annotations have a slight different encoding compared to the annotations inserted by the Java compiler. All the other components remain the same (i.e., we build on NG2C modifications). We now follow with a description of the most important aspects of POLM2's implementation.

5.4.2 Java Agents

Both the *Recorder* and *Instrumenter* are implemented as JVM TI agents. These are pluggable components that run attached to the JVM. Both agents (*Recorder* and *Instrumenter*) use ASM,[2] a Java bytecode manipulation library, to rewrite bytecode at load time. The *Recorder* uses ASM to add callbacks to the recording code on every object allocation while the *Instrumenter* uses NG2C API to direct object allocations to different generations.

In addition to recording allocations, the *Recorder* also serves a second purpose. It prepares the heap for a memory snapshot, and signals the *Dumper* component for creating a new snapshot. By default (this is configurable), upon each GC cycle finish, the *Recorder* instructs the *Dumper* to create a new snapshot. However, to avoid snapshotting memory pages that are not being used by the application and therefore to reduce the size of the snapshot, the *Recorder* traverses the Java heap and sets a special bit in the kernel page table (from here on called no-need bit) for all pages containing no live objects. This no-need bit that is set using the `madvise` system call, is used by the *Dumper* to avoid pages with the bit set (i.e., unnecessary pages) while creating the memory snapshot.

5.4.3 Efficient JVM Snapshots with CRIU

To perform JVM snapshots, POLM2 uses CRIU. CRIU allows any process to be checkpointed, i.e., all its resources (including its memory pages) are captured in a snapshot.

CRIU supports incremental checkpoints by relying on a kernel memory page table dirty bit that indicates if a particular page was dirtied or not, since the last snapshot. Upon each memory snapshot, CRIU cleans the bit and therefore, when a new snapshot is requested again, only pages with the dirty bit captured into the snapshot.

Finally, CRIU also ignores pages which are marked as not necessary by checking the no-need bit in the page table of a particular process. This no-need bit is set by the *Recorder* and is used to discard every page that contains no live objects.

[2] ASM is a bytecode manipulation library that is available at asm.ow2.org

Note that we do not introduce a new page bits or do any modification to the OS kernel. Instead, we use already existing bits, which are used for other applications and by the kernel.

By combining CRIU with the *Recorder*, POLM2 is able to create snapshots whose size (and consequently the time necessary to create them) is greatly reduced when compared to a usual approach using usual JVM tools such as `jmap`[3] (see results in Section 6.4).

5.4.4 Finding Recorded Objects in JVM Snapshots

Matching the objects' unique identifiers (object ids) recorded by the *Recorder* with the ids of the objects included in the JVM memory snapshot is a non-trivial implementation challenge. Remember that these object ids are used to match allocation records by the *Recorder* with the objects contained in the dumps created by the *Dumper*. In particular, simply using the `jmap` tool would not be possible because the ids of objects included in the heap dumps produced by `jmap` change over time as they are generated using their corresponding memory addresses. Since an object might be moved (promoted or compacted), its id might change, therefore breaking the goal of tracking the object until it becomes unreachable. Thus, in order to properly match the ids provided by the *Recorder* and the *Dumper*, another solution is used.

The objects' ids recorded by the *Recorder* are obtained by calling the method, available is OpenJDK, `System.identityHashCode` for each recorded object. Each id is generated by hashing the corresponding object and is stored internally in the JVM in the object's header. In order to successfully match object ids included in the snapshot, the *Analyzer* must read each object header in order to extract the object id and match it with the ids reported by the *Recorder*. Note, however, that it is possible that many objects recorded by the *Recorder* did not appear in the JVM snapshots since the *Analyzer* only traverses live objects.

5.4.5 Reducing Changes Between Generations

As discussed in Section 4.4.4, for each *STTree* leaf node (object allocation) that is not included in any conflict, there are two calls to the NG2C API: one to change the target generation of the leaf node, and one to go back to the previous target generation.

In order to reduce the overhead associated to calling NG2C multiple times, POLM2 avoids many of these calls by pushing the target generation to parent nodes in the hope that other leaf nodes in the same subtree also share the same target generation. Thus, if no conflicts are created, the target generation is set only when the execution enters into a specific subtree, and

[3] Java Memory Map, or `jmap`, is a tool for Java applications that can create application heap dumps.

all the leaf nodes included in the subtree inherit the target generation; thus, there is no need to call NG2C multiple times.

This optimization leads to a significant reduction in the number of calls to NG2C, therefore reducing the overhead associated with selecting the correct generation for object allocations. Note that this optimization is not applicable no sub-trees that contain conflicts since there will be different target generations.

5.4.6 Profiling Information for Generational GCs

As already mentioned, the overall goal of this work is to reduce Big Data application pause times by performing life time-aware memory management. To this end, POLM2 uses NG2C, a GC which exports an API to specify where objects should be allocated, decision that must take into consideration the expected life time of each object. However, it is important to note that POLM2 is completely independent of the GC that is being used. In other words, POLM2 can be used with any generational GC that supports pretenuring. The only code that would have to be changed is the GC-specific code in the *Instrumenter* (that specifies in which generation objects should be allocated in).

5.5 ROLP's Implementation

This section now describes a set of implementation details and performance optimizations that are essential to enable ROLP to succeed in a production JVM. To achieve its goals, ROLP needs to install profiling code and maintain several data structures. ROLP also integrates with NG2C, which takes advantage of the profiling information gathered by ROLP to automatically pretenure objects, i.e., it allocates objects with similar life times close to each other.

Since HotSpot is a highly optimized production JVM, new algorithms/techniques must be implemented carefully so as not to break the JVM's performance. This section describes some of ROLP's implementation details, in particular, the ones we believe to be important for realistically implementing ROLP in a production JVM.

5.5.1 Implementation Overview

ROLP is integrated with NG2C, the proposed pretenuring collector that allows the heap to be divided into an arbitrary number of generations (see Section 4.3). Figure 5.5 presents a graphical representation of the components (painted in green) that were introduced (Life Time Predictor and OLD Table) and changed (JIT) to implementt ROLP. The motivation behind NG2C

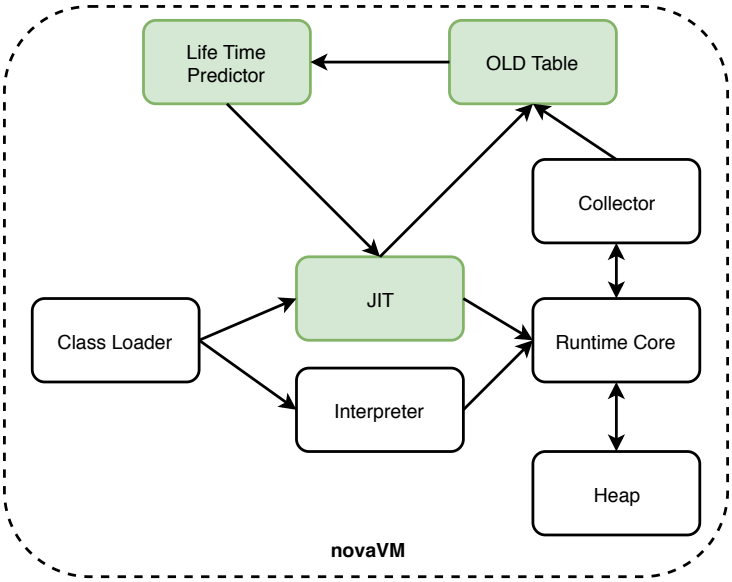

Figure 5.5: ROLP Implementation Components

is to be able to allocate objects with similar life times in the same generation. By doing so, objects will tend to die in groups, therefore reducing the fragmentation of the heap and leading to shorter pause times (when compared to G1).

In order to integrate ROLP with NG2C, we pre-configured NG2C to have 16 generations (the young generation, the old generation, and other 14 generations). In practice, what NG2C does is to sub-divide G1's old generation into multiple allocation spaces, and allow the collector to allocate application objects into each of these allocation spaces (dynamic generations). The number of generations (16) is used because it represents the maximum age of an object in HotSpot.

By default, NG2C relies on code annotations (hand-placed by the programmer, or introduced by POLM2) which are used during interpretation and JIT compilation to indicate in which generation/allocation space should an object be allocated. With ROLP, we modified NG2C not to look for code annotations but to use ROLP profiling results instead (coming from the Life Time Predictor). This profiling information is kept in table (OLD Table) which maintains the estimated age of objects for each allocation context (resulting from the analysis done in Section 4.5.4). Upon object allocation, we instruct NG2C to look into this table and to use the estimated age of an object (a number between 0 and 15) as the number of the generation to use (i.e., where that object will be allocated); e.g., in one hand, if the estimated age is zero, NG2C allocates the object in the young generation; on the other hand, if the estimated age is 4, NG2C allocates the object in one of the dynamic generations (generation 4).

For collecting garbage, NG2C relies on G1. The only code added by ROLP on top of NG2C is the code to update the Object Life Time Distribution table, when object survive a collection.

5.5.2 Dealing with Inlining, Exceptions, and Stack Replacement (OSR)

The HotSpot JVM is one of the most optimized runtimes and, therefore, we had to be extra careful to follow those optimizations and to ensure that we could get good performance. In this section, we analyze some techniques used by the JVM and how ROLP handles them.

Method Inlining

Method inlining is an important performance optimization for JVM applications. It allows a call to method *A* to be replaced with its code. This can lead to significant performance improvements as the cost of the `call` instruction is completely avoided.

There are a number of factors that control how the JIT compiler in HotSpot deals with inline methods such as the size of the method, and if the call in hand is polymorphic or not (i.e., if it can result in an invocation to different methods).

After studying this problem and analyzing both real application code and execution logs from JIT compilation, we realized that most methods being inlined contain very little control flow, and and are mostly simple operations that, because of being done many times, are abstracted into a separate method.

With this observation in mind, and trying to reduce the number of profiled method calls (to reduce the throughput impact of ROLP), we decided not to profile inlined method calls, i.e., whenever the JIT is inlining a method call (i.e., replacing the call with the actual method implementation), we do not include any profiling code to track the application context around the method that is being inlined. In addition, we conducted several experiments with an without this optimization (using the benchmarks used in Section 6.5) and we noticed that no conflict was unresolved after using this optimization.

Exception Handling

Exception handling is another important topic as it breaks the assumption that after returning from a method, the stack state of the executing thread will be updated (remember that we update the threads' stack state before and after each method call). However, exceptions can break this technique as an unhandled exception will climb the stack until: i) there is a suitable exception handler, or ii) the application terminates with the exception.

In practice, when an exception is thrown, the JVM will look for a suitable exception handler to handle it. If there is no suitable handler in the current method, the exception is automatically re-thrown, and is going to be catched by the JVM stubs in the caller method. Note that when the

JVM re-throws an exception, the execution goes directly to the JVM stub in the caller method, i.e., the profiling code installed right after the call is not executed.

In order to fix this problem, and to avoid the thread-local stack state being inconsistent with the stack state, ROLP hooks the code to update the stack state whenever the JVM decides to re-throw an unhandled exception. This way, even if exceptions are not handled in the current method, exiting a method through an unhandled exception will not lead to a corruption of the stack state.

On Stack Replacement

On Stack Replacement (OSR) is yet another important technique used by HotSpot JVM. This technique allows the JVM to switch how a particular method is executed (either through interpretation or through some JIT compiled code) while the method is being executed. This technique is particularly useful in two situations. First, if the method being executed is not called very often but takes a long time to execute (e.g., executing a loop), the JVM can still JIT compile the method and replace the stack frame with the exact equivalent version. In other words, the JVM can still optimize code (by jitting it) even if the method is still executing. Second, it is also possible to use this technique to de-optimize methods; e.g., if a different unpredicted branch is taken and the JVM needs to re-compile the method again.

In short, OSR can also be harmful for ROLP's stack context updates because any method in the stack can go from an interpreted method into a compiled method. Given that with ROLP we only install profiling code in jitted code, switching implementations after executing a particular method would corrupt the stack context.

To solve this problem, we periodically verify the correctness of the application threads' context. This is done at the end of each GC cycle, while all application threads are still stopped. If ROLP finds an incorrect context state, it will correct its value, making it consistent with the real execution stack. After testing the performance of applications with and without this technique, we concluded that its cost is negligible, and is absorbed by the cost of the other collection tasks.

5.5.3 Reducing Profiling Overhead for Very Large Applications

Profiling large-scale applications can be challenging from the performance point of view. As shown in Section 6.5, even for DaCapo benchmarks with no context conflicts, some benchmarks experienced more than 10 % throughput overhead. In other words, even with highly optimized produced JIT code for profiling the application, it is not possible to reduce the throughput to negligible values for some applications.

To further reduce the throughput overhead, ROLP allows the definition of package-based filters to either profile or not profile a package (and all its sub-packages). We found this extremely useful and effective to bound the throughput overhead. In practice, we used this technique in the large-scale workloads (described in the Section 6.5) to focus the profiling effort on packages that manage application data structures. In addition, identifying these packages is almost effortless for most programmers.

5.5.4 Shutting Down Survivor Tracking to Reduce Application Pause Times

As mentioned above, helping pretenuring GCs to reduce pause times is one of the main motivations for ROLP. However, in our experiments, we observed that introducing profiling code to track object life times and allocation contexts can introduce some overheads. Despite the small throughput overhead, applications can experience increased GC pause times even if the amount of objects being copied is reduced.

After analyzing this strange effect, we realized that ROLP was introducing such a GC pause time overhead during the object survivor processing phase; this was due to the profiling code that extracts the allocation context from an object's header, and looks it up in the Object Life Time Distribution table. This operation is performed for every object that survives a collection. Thus, we noticed that, after starting to pretenure objects (using NG2C), the dominating phase of a GC cycle was the survivor processing phase.

Therefore, to further reduce the application pause times, ROLP can dynamically turn off the survivor tracking code. By doing this, it is possible to reduce even further GC pause times. Note that ROLP only performs this optimization (i.e., turning off the survivor tracking code) if the workload is stable (i.e., the profiling decisions regarding the estimated life time of objects did not change in the last iteration). Obviously, it is also possible to turn on the survivor tracking code again. Currently, this code is only turned back on if the average pause time increases over 10% (this is a configurable value) compared to the last recorded value when the survivor tracking code was active.

5.5.5 Object Life Time Distribution Table Scalability

ROLP uses a global table (Object Life Time Distribution) which is accessed very frequently. In order to provide average constant time for insertion and search, this data structure is implemented as a hashtable.

Another important concern is how large is the memory budget to hold this table in memory. In the worst-case scenario, and since the allocation context is a 32 bit value, one could end up

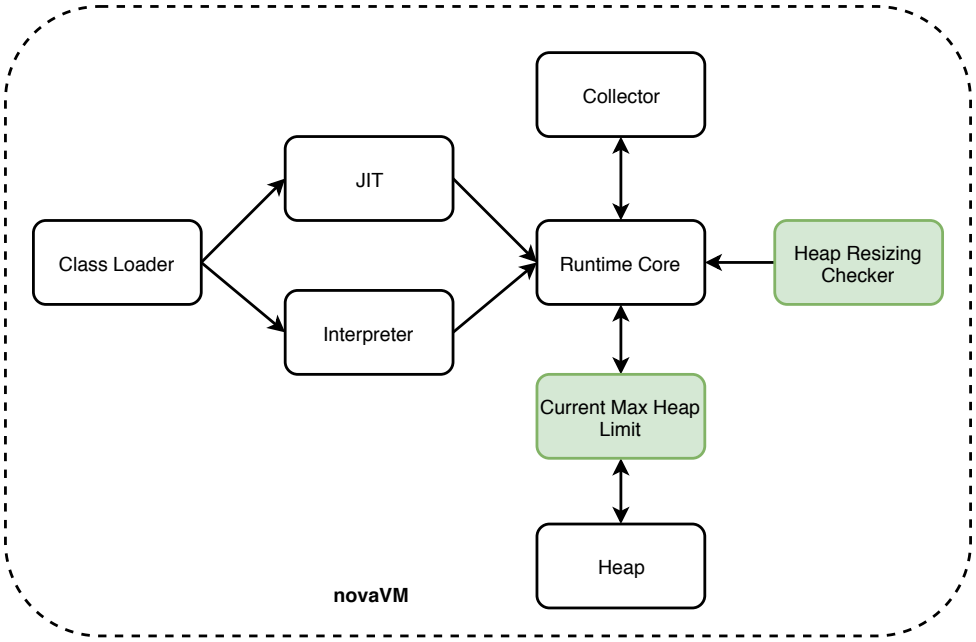

Figure 5.6: Dynamic Vertical Scalability Implementation Components

with a table with 2^{32} entries which would take 4 bytes * 16 columns * 2^{32} entries (approximately 256 GBs). However, in practice, we are able to keep the size of this table to a much lower value (as can be see in the Section 6.5).

Initially, the table is initialized with 2^{16} entries, one for each possible allocation site identifier. At this point, the table occupies approximately 4 MB of memory. Whenever a conflict is detected, the table size is increased by 2^{16} to be able to accommodate all possible stack state values for the specific allocation site where the conflict was found. Hence, the size of the table is 2^{16} * (1 + *N*) entries, which is equivalent to 4 * (1 + *N*) MB, where *N* is the number of detected conflicts.

5.6 Vertical Scaling Implementation

The proposed ideas in Section 4.6 were implemented as a new component of novaVM. In addition, we currently support two widely used OpenJDK collectors: i) Parallel Scavenge (PS), and ii) Garbage First (G1). Note that, since NG2C is implemented on top of G1, NG2C is also supported. Thus, in this section, when G1 is mentioned, NG2C is also applicable.

5.6.1 Implementation Overview

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 work. In particular, as de-

picted in Figure 5.6, we had to introduce two components: the Heap Resizing Checker (further described in Section 5.6.3) and the Current Max Heap Limit (further described in Section 5.6.2). 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 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).

5.6.2 Dynamic Memory Limit

As previously discussed in Section 2.5, the JVM allows the user to specify, at launch time, a number of configuration parameters, one of which, `MaxMemory` (the maximum memory limit). This limit is static and therefore cannot be changed at runtime.

To implement a dynamic memory limit (i.e., a memory limit that can be changed at runtime), we create a new JVM runtime variable, called `CurrentMaxMemory`, which can be set at launch time using either the JVM launch arguments and/or changed at runtime using an OpenJDK tool named `jstat`. Since this variable must respect the invariant presented in Section 4.6.1, every time a new value is requested, the JVM executes the code presented in Algorithm 6.

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.

5.6.3 Heap Resizing Checks

As discussed in Section 2.5, it is also necessary to timely reduce the heap size (`committed` memory) to return unused memory back to the host engine. To do so, the code presented in Algorithm 7 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 to 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.

5.6.4 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 in 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 7) returns true and forces a heap resizing.

5.7 Summary

This chapter presented and explained the implementation of novaVM and how all the subsystems, implementations of the algorithms presented in the previous chapter, integrate together, forming an enhanced JVM for Big Data applications.

We would like to emphasize that all the source code is opensource and is accessible at `github.com/rodrigo-bruno`. In addition, as discussed in Chapter 1, multiple contributions to the opensource community were made during the development of novaVM. In particular, both CRIU and the OpenJDK HotSpot projects have received and accepted code contributions from this thesis, further showing the usefulness of the work developed and its relevance.

Chapter 6

Evaluation

Having analyzed the architecture and implementation of novaVM, this chapter focuses on evaluating novaVM. Thus, the goal of this chapter is to assess if the proposed problems are solved while complying with the proposed requirements (as presented back in Chapter 1). This evaluation is of special relevance because novaVM builds on top of a highly optimized system (OpenJDK HotSpot), whose performance is critical for many industrial applications. Therefore, any performance and/or user-experience degradation should be carefully analyzed.

The evaluation is divided into three main parts, one for each of the problems presented in Chapter 1. Hence, the performance of ALMA (Section 4.2) is evaluated in Section 6.2. NG2C (Section 4.3), POLM2 (Section 4.4), and ROLP (Section 4.5) are evaluated in Sections 6.3, 6.4, and 6.5, respectively. Note that although these solutions provide solution to Problem 2, we study them in a separate section to allow a deeper look into specific details of each subsystem. However, we also show their performance side-by-side in Section 6.5 to allow a better comparison of the different possible approaches. Finally, the performance of our Dynamic Vertical Scaling sub-system (Section 4.6) is presented in Section 6.6.

Despite the fact the we evaluate the solution to each problem independently, the proposed JVM contains all proposed algorithms all the time. We do evaluate each solution separately to allow a more precise comparison with previous work and to easily identify performance benefits and degradations coming from each algorithm.

In the following section, we describe the benchmarks and workloads used to evaluate novaVM. In this work we use a combination of real-world and synthetic benchmarks and workloads, whose goal is to approximate real-world scenarios. These are used in several experiments and we will explicitly indicate which of the benchmarks and workloads are used for each experiment and if there was or not any modification (along with the reason for the modification) with regards to the original benchmark and/or workload.

Workload	Short Description
avrora	simulates programs ran on a grid of AVR microcontrollers
eclipse	executes jdt performance tests for the Eclipse IDE
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file
h2	executes a JDBC in-memory benchmark
jython	inteprets the pybench Python benchmark
luindex	uses lucene to index a set of documents
lusearch	uses lucene to do a text search of keywords
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a set of images using ray tracing
tomcat	runs a set of queries against a Tomcat server
tradebeans	ray tracer benchmark via a Jave Beans to a GERONIMO backend
tradesoap	ray tracer benchmark via a SOAP to a GERONIMO backend
xalan	transforms XML documents into HTML

Table 6.1: DaCapo Benchmarks Summary

Workload	Short Description
scimark	floating-point operations benchmark
derby	real-world inspired database benchmark
crypto	crypto operations benchmark
compress	data compression and decompression benchmark
xml	XML processing and validation benchmark
serial	serialization and deserialization of objects from JBoss
mpegaudio	MP3 encoding and decoding workload (heavy use of floating-point)

Table 6.2: SPECjvm2008 Benchmarks Summary

6.1 Workload Description

Throughout the evaluation we use a mixture of real-world applications and data, combined with real-world-based benchmarks. The decision of what workloads to use was mainly driven by three factors: i) real-world applications and data provide high fidelity to the evaluation results; ii) using widely used and studied benchmark suites containing many different workloads allows us to determine how the proposed algorithm is performs in many scenarios and allows easier analysis with already well-studied workloads; iii) using workloads used by previous works allows us to compare the proposed algorithms with previous solutions.

With the aforementioned factors in mind, we selected a number of workloads which we describe in this section.

6.1.1 DaCapo and SPECjvm Benchmark Suites

DaCapo [13] and SPECjvm [106] are two well-known and widely used benchmarks to analyze the performance of JVM implementations. Benchmarks included in these suites mimic real-world workloads with different performance aspects such as non-trivial memory intensive

workloads, CPU intensive workloads, among others. There are three main advantages of using such suites: i) we test our algorithms with many workload types; ii) the workloads used in these benchmarks are well studied facilitating the task of understanding potential god or bad results; and iii) it is easier to compare with other works that also use the same benchmarks.

A short description of all the benchmarks used is presented in Tables 6.1 (DaCapo) and 6.2 (SPECjvm). In most experiments, we tend to use DaCapo 9.12 more often because it is a more update suite compared to SPECjvm2008. The latter is mainly used to compare the performance of our approach with results from previous works.

6.1.2 Cassandra

Cassandra [79] is a very popular large-scale Key-Value store. We use this Big Data platform as an example of a storage platform. In our evaluation Cassandra is executed under 4 different workloads: i) Feedzai's workload (consisting of 500 read queries and 25000 write queries per second, for the whole Cassandra cluster); ii) write intensive workload (2500 read queries and 7500 write queries per second); iii) read-write workload (5000 read queries and 5000 write queries per second); iv) read intensive workload (7500 read queries and 2500 write queries per second).

Note that Feedzai's workload is based on anonymized data from real deployments of their product (i.e., credit card fraud detection). All workloads besides Feedzai's are synthetic but mirror real-world settings (we use the YCSB benchmark tool).[1]

6.1.3 Lucene

We use Lucene [90] to build an in-memory text index using a Wikipedia dump from 2012.[2] The dump has 31 GB and is divided in 33M documents. Each document is loaded into Lucene and can be searched. Lucene is used as yet another example of a storage platform, which can sustain read and write requests from users.

The workload is composed by 20000 writes (document updates) and 5000 reads (document searches) per second; note that this is a write intensive workload which represents a worst case scenario for GC pauses. For reads (document queries), we loop through the 500 top words in the dump, the ones that will have more results (worst case).

[1] The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source benchmarking tool often used to compare NoSQL database systems.

[2] Wikipedia dumps are available at dumps.wikimedia.org

6.1.4 GraphChi

When compared to the previous systems (Cassandra and Lucene), GraphChi [78] is a more throughput oriented system (and not latency oriented). Therefore, it is used as an example of a processing platform.

We use GraphChi for two reasons: i) we want to demonstrate that novaVM does not degrade throughput even in a throughput oriented system; ii) with novaVM, systems such as GraphChi can now be used for applications providing latency oriented services, besides only performing throughput oriented graph computations.

In our evaluation, we use two well-known algorithms: i) page rank, and ii) connected components. Both algorithms are feed with a 2010 Twitter graph [77] consisting of 42 millions vertexes and 1.5 billions edges. These vertexes (and the corresponding edges) are loaded in batches into memory. GraphChi calculates a memory budget to determine the number of edges to load into memory before the next batch. This represents an iterative process; in each iteration a new batch of vertexes is loaded and processed.

6.1.5 Tomcat

Finally, we use the Tomcat web server [3] which allows us to mimic real-world usage of websites in the cloud. The workload used with Tomcat is based on real-world utilization of Tomcat in the Jelastic cloud. We use this platform to demonstrate the performance of our Dynamic Vertical Scaling sub-system.

In the following sections we proceed with the explanation and discussion of the evaluation results. Although the experiments will be using the workloads described in this section, in each section, we will clarify what is the environment where these workloads ran and if there was any change to the workload and/benchmark.

6.2 ALMA's Evaluation

This section describes the evaluation of ALMA for which we use SPECjvm2008 and DaCapo 9.12 benchmark suites to exercise our solution. These benchmark suites are specially interesting as they contain different workloads that simulate multiple applications, allowing us to verify the performance of our approach when migrating different types of applications (for example, memory intensive or CPU intensive).

ALMA is compared against the following solutions:

[3] Apache Tomcat is a Java Servlet Container. It can be reached at tomcat.apache.org.

- **CRIU** [71] (checkpoint and restore tool for Linux processes, presented in Section 3.1.1); this solution uses NFS to transfer snapshots from the source site to the destination site; besides, it does not take into consideration unused or unreachable memory and therefore, snapshots all memory allocated to a particular process;

- **JAVMM** [60] (presented in Section 3.1.1), a recent system with the same goal as ALMA: 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 et al. [60].[4] 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 (due to not having access to the source code);

- **ALMA-PS** which is the ALMA solution using the GC and tuning proposed in JAVMM [60] (this solution was prepared only for the purpose of this evaluation); in other words, ALMA-PS uses the Parallel Scavenge GC with 1 GB for young generation and 1 GB for old generation, and forces one minor (young) collection upon snapshot creation (settings described in Hou et al. [60]). We use this system to: i) isolate the performance benefits of using JVM migration versus system-VM migration 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 1 GB 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 4.2.1 for details).

[4] 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.

In this section we start by describing the evaluation environment and by characterizing the applications included in both benchmarks suites with regards to memory utilization, and then present the evaluation results for: 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.

These experiments assess the efficiency of ALMA for the three main performance metrics (regarding live migration systems): i) total migration time, ii) application overhead, and iii) resources overhead. The last experiment provides insights about the performance expectations when using more processing power (more cores) and/or more network bandwidth.

6.2.1 Evaluation Environment

All the three solutions (ALMA, ALMA-PS, and CRIU) were executed on a local OpenStack[5] 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 40 GB 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 2 GBs of RAM except when we run experiments with ALMA-PS, which needs 4 GBs of RAM to run (more details in Section 6.2.3).

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) 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 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 runs faster when a migration needs to be performed.

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

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 6.3: Benchmark Analysis for SPEC (above) and DaCapo (below)

6.2.2 Benchmark Characterization

Table 6.3 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 they could not run in our JVM (as they fail to compile; for example the compiler benchmark application from SPEC) or could not be migrated using CRIU (for example the tradebeans and tradesoap benchmark applications from DaCapo).

The top rows (Table 6.3) 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 [6] (% 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 GC 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[7] GCs before the migration starts.

[6] 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.

[7] An allocation failure happens when no more free memory exists to satisfy an allocation request. This event

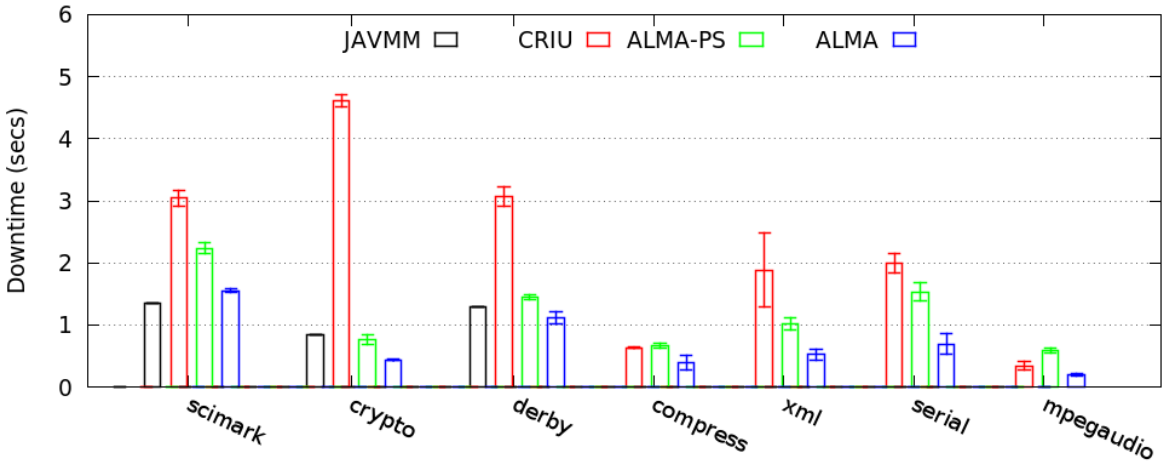

Figure 6.1: Application Donwtime (seconds) for SPECjvm2008 Benchmarks

We take the heap usage after the first GC and the heap usage right before the second one and divide it by the time elapsed between the two GCs. All values are averages of at least 5 runs (which we found to be enough to extract reliable statistical results for these benchmarks). This is also true for all values presented during this 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 6.3 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 6.3 also apply.

6.2.3 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 each system (ALMA, ALMA-PS, and CRIU), using a total of 15 applications (presented in Table 6.3). 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

triggers a garbage collection.

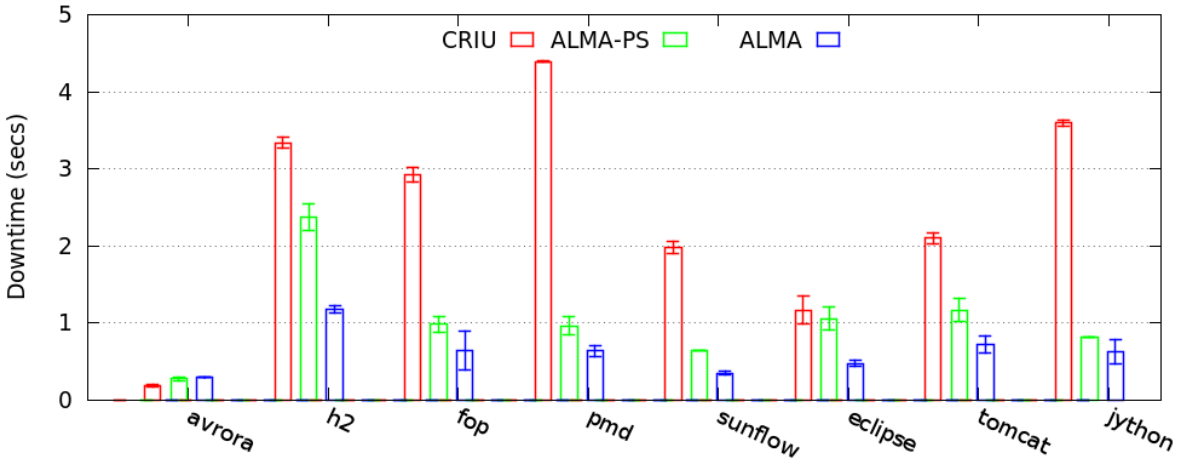

Figure 6.2: Application Donwtime (seconds) for DaCapo Benchmarks

their maximal resource consumption (mainly CPU and memory). All applications run five times during which we calculate both the average and the standard deviation for these runs.

Figures 6.1 and 6.2 show the results for the application downtime. For each benchmark application, results are grouped, having one bar for each system (from left to right): JAVMM, CRIU, ALMA-PS, and ALMA. This organization of columns is also used in subsequent figures.

CRIU is the migration solution with worse downtime. This is because it snapshots all the process memory, not taking into account 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 mpegaudio) but still worse than ALMA. The reasons are the following. First, ALMA-PS initializes the young generation with the size of 1 GB. This forces the migration solution to handle a snapshot of 1 GB 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 24 MB of memory (Table 6.3). Therefore, the overhead of handling 1 GB instead of approximately 24 MB makes ALMA-PS perform worse than ALMA and CRIU. In ALMA, we do not impose such young generation size pre-condition 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 6.1), ALMA achieves better results in 2 out of 3 applications. Considering that the environment in which both systems

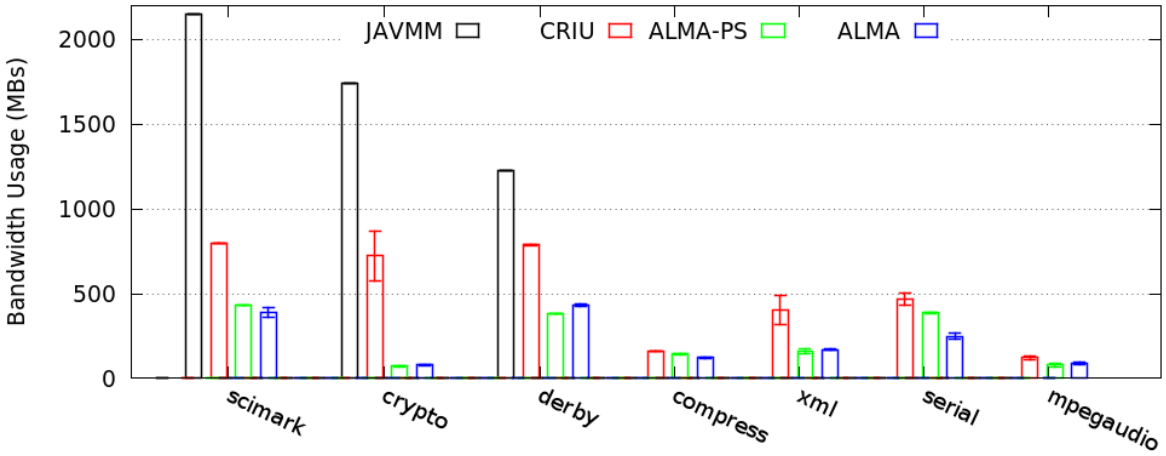

Figure 6.3: Network Bandwidth Usage (MBs) for SPECjvm2008 Benchmarks

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 that will be migrated. 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 6.3, 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 worst cases, the downtime with ALMA is less than CRIU, and ALMA-PS. ALMA and JAVMM achieve similar downtime results for scimark and derby.

6.2.4 Network Bandwidth Usage

We now 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 Figures 6.3 and 6.4). JAVMM clearly yields the worse results, even worse than CRIU. This is due to the fact

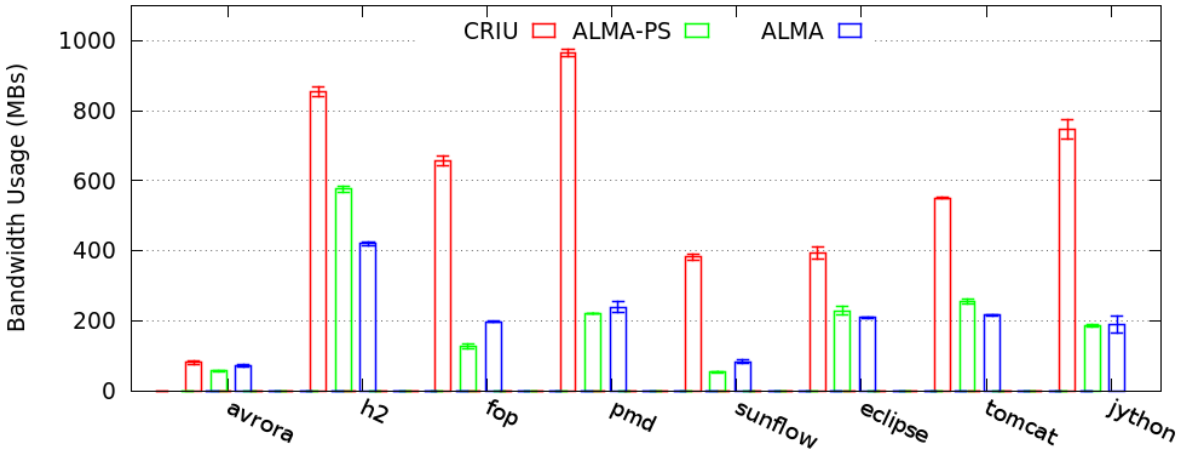

Figure 6.4: Network Bandwidth Usage (MBs) for DaCapo Benchmarks

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 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 many more inter-region references and, consequently, need more GC effort to collect the young generation; such inter-region references do not exist in the GC of ALMA-PS as all young objects are in the same space (young generation).

In general, applications that use more memory tend to consume more memory bandwidth during migration. From Table 6.3 and Figures 6.3 and 6.4, we may conclude that applications with more heap usage and less garbage percentage (e.g. scimark, derby, h2) result in increased network bandwidth usage.

6.2.5 Application Throughput

Figures 6.5 and 6.6 show the normalized results for the throughput of applications when using CRIU, ALMA-PS, and ALMA. These results are obtained by sampling the benchmark throughput (number of operations) during five seconds. We experimentally determined that five sec-

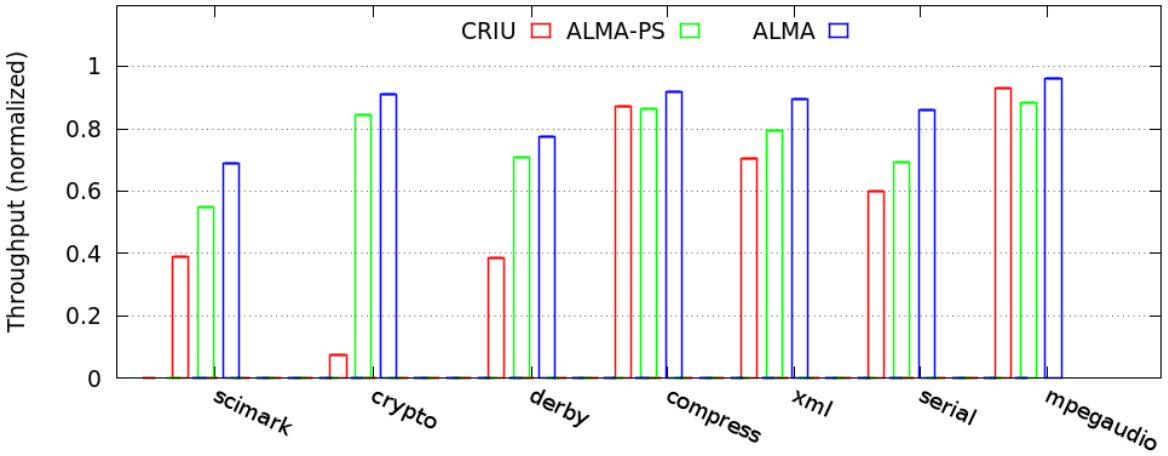

Figure 6.5: Application Throughput (normalized) for SPECjvm2008 Benchmarks

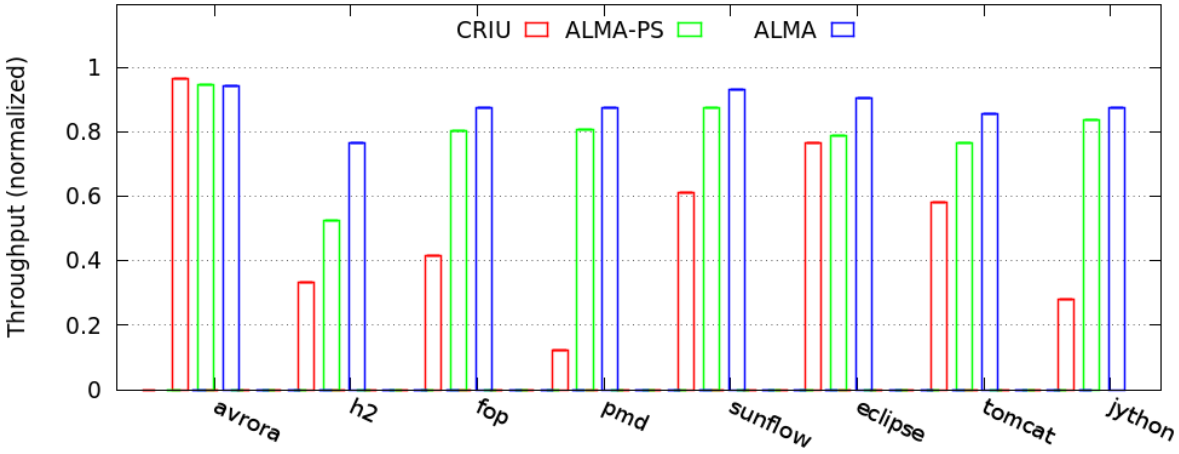

Figure 6.6: Application Throughput (normalized) for DaCapo Benchmarks

onds was enough to precisely measure the throughput of each application during a migration. In addition, five seconds allow each benchmark to execute at least one iteration. Note that we measure during the time interval that includes the migration. 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 Figures 6.5 and 6.6. The normalized throughput for each system, represents the throughput achieved when the migration occurred.

The throughput results have a strong correlation with the application downtime (see Section 6.2.3). 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

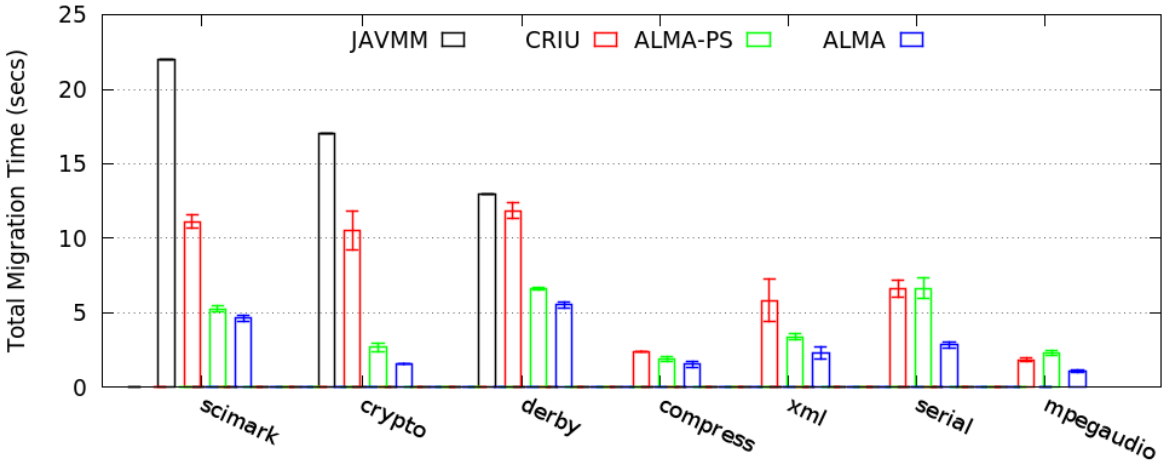

Figure 6.7: Total Migration Time (seconds) for SPECjvm2008 Benchmarks

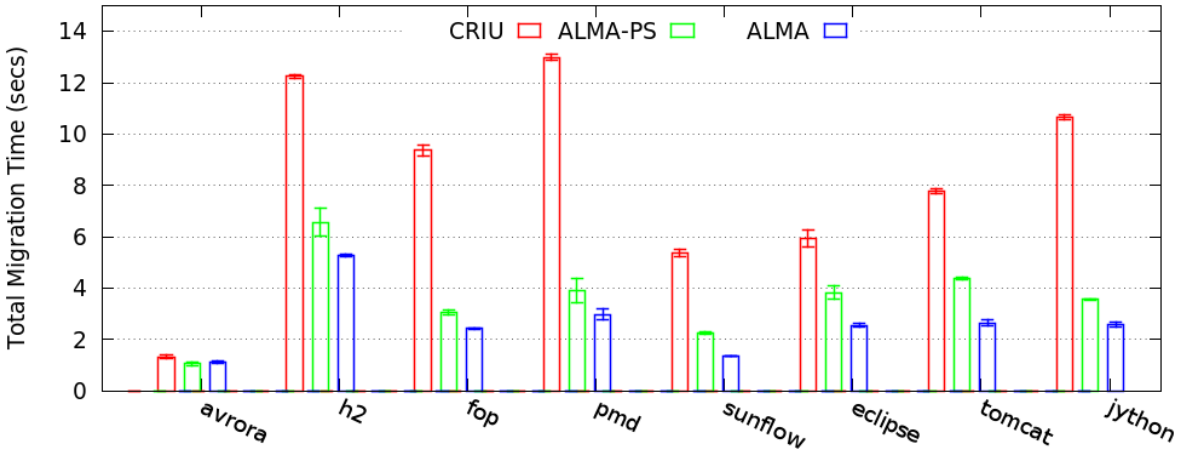

Figure 6.8: Total Migration Time (seconds) for DaCapo Benchmarks

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 benchmark applications.

6.2.6 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.

From Figures 6.7 and 6.8, we can see that JAVMM performs worse than all others. This

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 6.4: Performance Results Normalized to ALMA

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 Section 6.2.4 (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 in some cases, has the drawback of forcing a 1 GB 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 1 GB.

Table 6.4 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 (no source code access).

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.

ALMA-PS presents the closest performance results when compared to ALMA. Table 6.4 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).

6.2.7 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 4.2.1). This section shows the performance penalty of running such a migration-aware GC. Table 6.5 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

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 6.5: ALMA Migration Aware GC Overhead Compared to G1 GC for SPEC (above) and DaCapo (below)

Migr. GC - using the migration aware policy described in Section 4.2.1), 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 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 three 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[8] 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.

6.2.8 ALMA with More Resources

In this last experiment, we study the performance impact of ALMA on the application downtime while increasing the number of cores used by the application and the network bandwidth avail-

[8] 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.

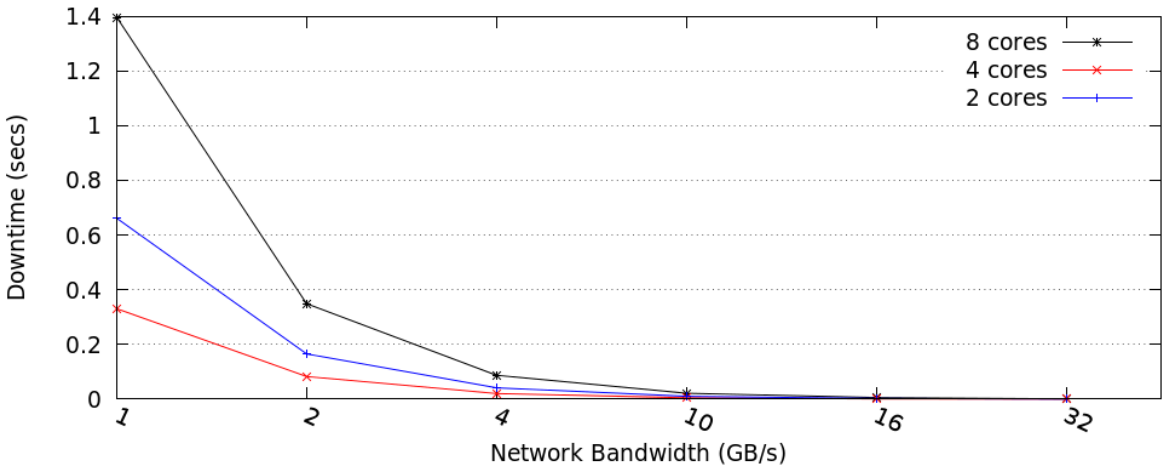

Figure 6.9: ALMA Application Downtime With More Cores Versus More Network Bandwidth

able (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 (all used by the benchmark application). 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.

$$Downtime = \frac{SizeIncSnapshot}{NetBandwidth} \tag{6.1}$$

$$SizeIncSnapshot = \frac{SizeInitSnapshot}{NetBandwidth} * DirtyRate \tag{6.2}$$

$$Donwtime = \frac{SizeInitSnapshot}{NetBandwidth^2} * DirtyRate \tag{6.3}$$

The results are shown in Figure 6.9. Since our installation only has 1 Gbps, we estimate the remaining values with more network bandwidth. The estimated values are obtained as follows (see Eqs. 6.1, 6.2, and 6.3): 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 by the dirty rate (assuming it is constant), we get the size of the incremental snapshot (Eq. 6.2). This enables us to estimate the application downtime (by replacing Eq. 6.2 in Eq. 6.1 to obtain Eq. 6.3). We also consider that part of both the initial and the incremental snapshots are filtered as garbage (this percentage is taken from Table 6.3).

Figure 6.9 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).[9] One important conclusion to take from our experiment is that the application downtime (see Eq. 6.3) 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). The same conclusions could, however, be extended to an arbitrary number of snapshots.

6.3 NG2C's Evaluation

We now dedicate this section to evaluate the performance of NG2C. We compare NG2C's performance with G1 (Garbage-First, see Section 3.2.2), CMS (Concurrent Mark-Sweep, see Section 2.4), and C4 (Completely Concurrent Compacting Collector, see Section 3.2.2). Although not being an OpenJDK collector, C4 comes from a similar JVM, Zing.[10] Since we only have one license available, we only run a limited number of experiments experiments with Zing JVM (we clearly indicate in which experiments we show results for C4).

For evaluating NG2C, we do not use either SPECjvm or DaCapo as these benchmark suites are mostly optimized to evaluate the JVM in terms of throughput and memory footprint but not in terms of latency. Thus, we use three relevant platforms that are used in large-scale environments: i) Apache Cassandra 2.1.8 [79], a large-scale Key-Value store, ii) Apache Lucene 6.1.0 [90], a high performance text search engine, and iii) GraphChi 0.2.2 [78], a large-scale graph computation engine. Each was this platforms and workloads used were described in Section 6.1.

For evaluating NG2C, we are mostly concerned on showing that, compared with other collectors, NG2C: i) does reduce application pause times; ii) does not have a negative effect neither on throughput nor on memory utilization; iii) greatly reduces object copying; iv) does not increase the remembered set management work.

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

[10] Zing is a JVM developed by Azul Systems (www.azul.com).

Workload	CPU	RAM	OS	Heap	*Young*	LOC
Feedzai	Intel Xeon E5-2680	64 GB	CentOS 6.7	30 GB	4 GB	22
WI,RW,RI	Intel Xeon E5505	16 GB	Linux 3.13	12 GB	2 GB	22
Lucene	AMD Opteron 6168	128 GB	Linux 3.16	120 GB	2 GB	8
PR,CC	AMD Opteron 6168	128 GB	Linux 3.16	120 GB	6 GB	9

Table 6.6: Evaluation Environment Summary

6.3.1 Evaluation Environment

We evaluate NG2C in three different environments (Table 6.6 provides a summary of the evaluation environments). First, we use Feedzai's internal benchmark environment. This environment mirrors a real-world deployment and uses a Cassandra cluster to store data. For Feedzai, it is very important to keep Cassandra's GC pauses as short as possible to guarantee that client SLAs are not broken by long query lantencies. The Cassandra cluster is composed by 5 nodes.

Second, we use a separate node to evaluate NG2C with Cassandra under three different synthetic workloads with varying number of read and write operations (as described in Section 6.1): Write-Intensive (WI), Write-Read (WR) and Read-Intensive (RI).

Given the size of the data sets used for Lucene (Wikipedia dump) and GraphChi (Twitter graph dump), we use another separate node to evaluate these platforms with NG2C. On top of Lucene we perform client searches while continuously updating the index (read and write transactions). For GraphChi, we use two workloads, PageRank and Connected Components.

Each experiment runs in complete isolation for at least 5 times (i.e., until the results obtained become stable). Feedzai's workload runs for 6 hours, while all other workloads run for 30 minutes each. When running each experiment, we never consider the first minute of execution (in Feedzai's benchmarks we disregard the first hour of execution to allow other external systems to converge). This ensures minimal interference from JVM loading, JIT compilation, etc.

We always use fixed heap and *Young* generation sizes (see Table 6.6). We found that these sizes are enough to hold the working set in memory and to avoid premature massive promotion of objects to older generations (in the case of CMS and G1). Table 6.6 also reports the number of lines changed for taking advantage of NG2C (LOC column).

6.3.2 NG2C Platform Code Changes

As discussed in Section 4.3, NG2C requires code changes so that the developer can tell NG2C which allocation sites should be used for pretenuring and how. Therefore, in this section we illustrate the changes that we had to do in each platform's source code in order to take advantage of NG2C.

Cassandra

To use NG2C we modified Cassandra code to mainly allocate all objects belonging to a particular `Memtable`[11] in a separate dynamic generation. Thus, whenever a new `Memtable` is created or flushed, we create a new dynamic generation. Each `Memtable` contains a B-Tree (self-balancing tree data structure) with millions of objects. These objects contain references to buffers with real data. To take advantage of NG2C, we allocate all objects and buffers belonging to a particular `Memtable` in the dynamic generation created for that specific `Memtable`.

In total, we changed a total of 22 code locations: i) 11 code locations where we annotate the `new` instruction, and ii) 11 code locations where we create, or change generation.

Lucene

To reduce Lucene's GC pauses the code was mainly modified to allocate documents' data (of the Wikipedia dump) in the *Old* generation. Objects created to hold the indexes of documents will live throughout the application life time; therefore, if we do not use NG2C such objects would be copied within the heap (thus leading to long GC pauses). With NG2C, most objects holding the index (including objects such as `Term`, `RAMFile` and `byte` buffers) are allocated outside the *Young* generation. To accomplish it, we changed 8 code locations in Lucene, all of which to annotate the `new` instruction.

GraphChi

To take advantage of NG2C, we changed GraphChi in several code locations. The code was mainly modified to allocate objects representing graph vertexes (`ChiVertex`), edges (`Edge`), and internal pointers (`ChiPointer`) in multiple dynamic generations. GraphChi splits the overall graph computation in batches (i.e. group of nodes and edges to process). We use a dynamic generation for each separate processing batch. We modified a total of 9 code locations, 8 of which used to annotate the `new` instruction.

6.3.3 GC Pause Times

Figures 6.10 to 6.16 present the GC pause Times for the different application and workloads. For each plot, we show results for CMS, G1, and NG2C, across multiple percentiles. We do not show pause times for C4 because it is a concurrent collector and therefore, the application should never be paused. In practice, using C4, we got pauses of only up to 15 milliseconds

[11] A `Memtable` table caches recent writes in memory. When a `Memtable` is full, a flush is scheduled and a new `Memtable` is created. The capacity of each `Memtable` is proportional to the JVM heap size.

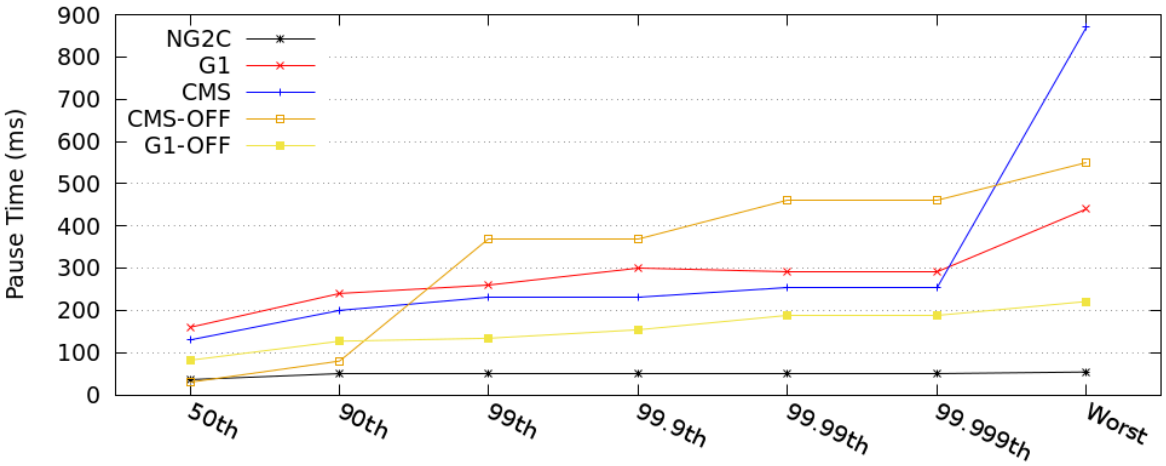

Figure 6.10: Pause Time Percentiles (ms) for Cassandra WI Workload

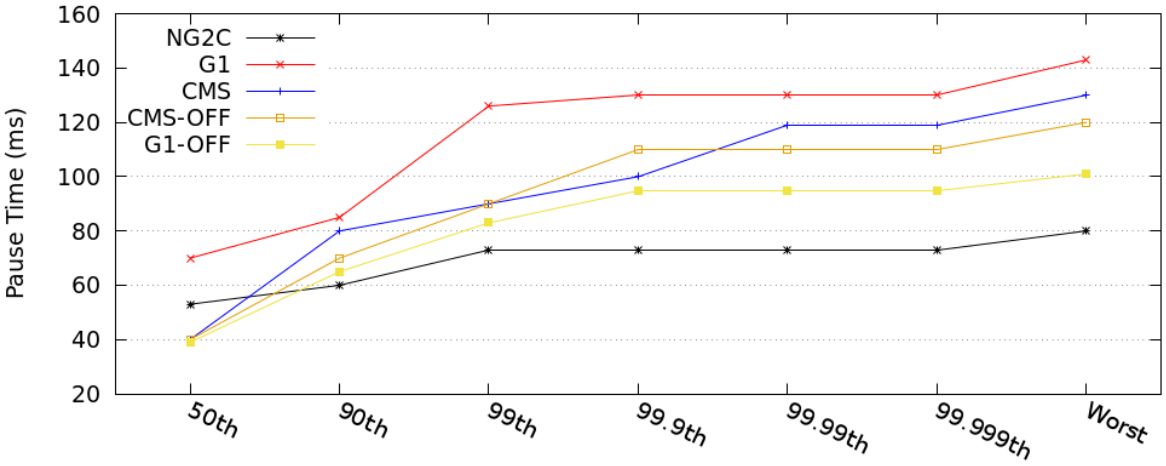

Figure 6.11: Pause Time Percentiles (ms) for Cassandra WR Workload

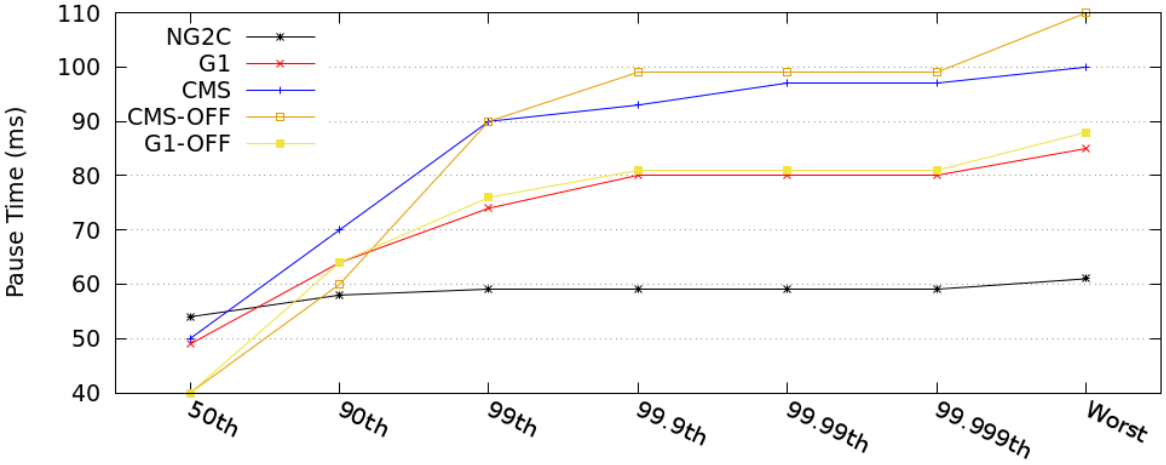

Figure 6.12: Pause Time Percentiles (ms) for Cassandra RI Workload

for Cassandra. However, as we see in Section 6.3.6, these short pauses come at the cost of reduced throughput.

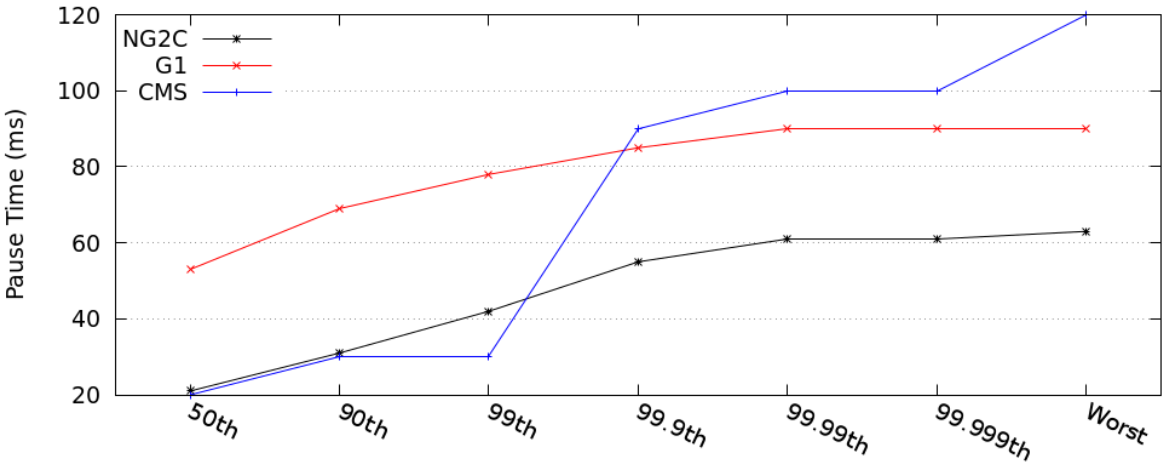

Figure 6.13: Pause Time Percentiles (ms) for Cassandra Feedzai Workload

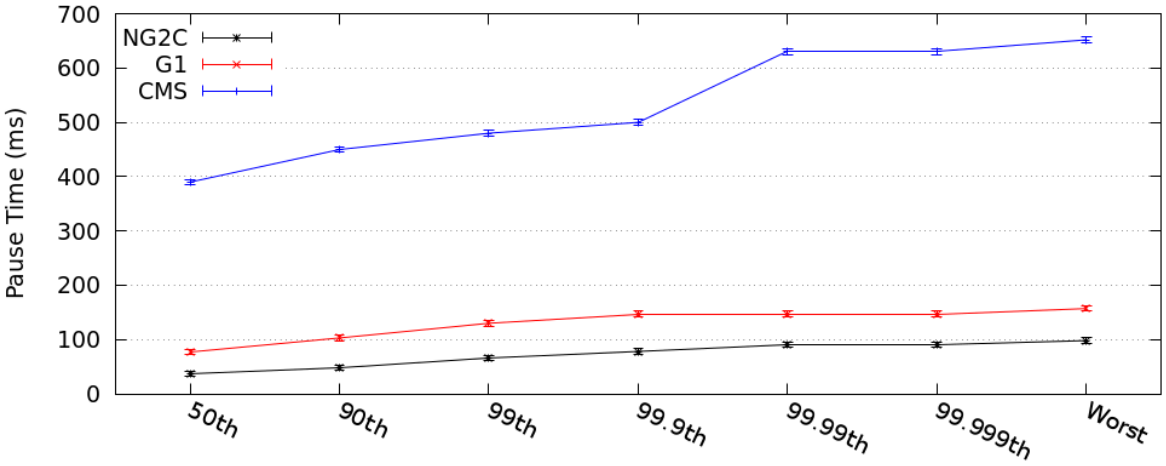

Figure 6.14: Pause Time Percentiles (ms) for Lucene Workload

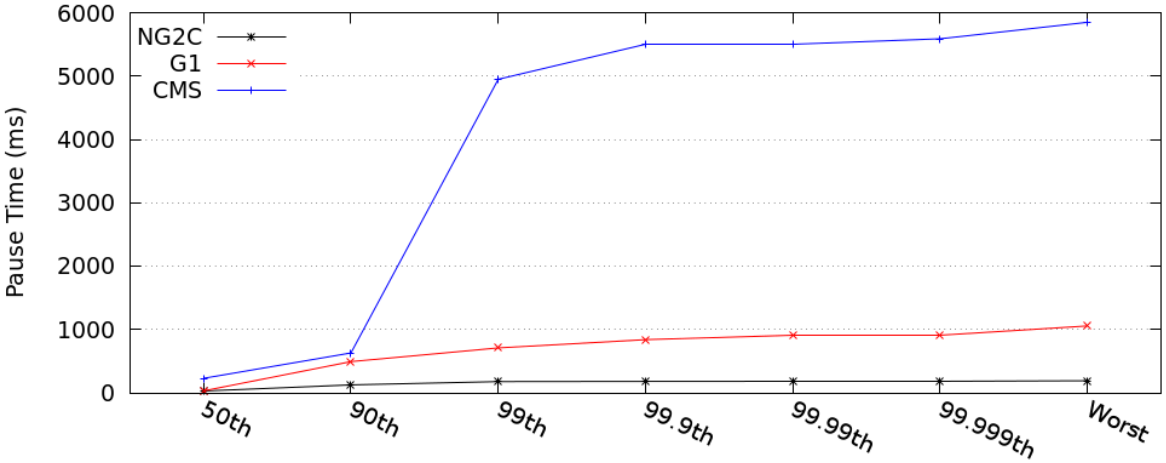

Figure 6.15: Pause Time Percentiles (ms) for GraphChi CC Workload

In Feedzai's workload (Figure 6.13), GC pauses are shorter when compared to the other Cassandra workloads. This is mainly because the hardware used in Feedzai achieves better

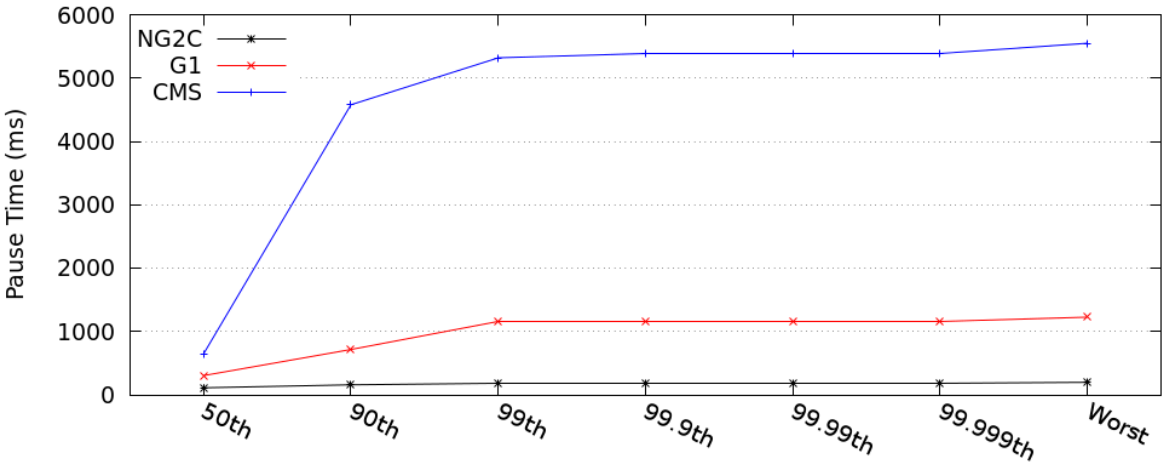

Figure 6.16: Pause Time Percentiles (ms) for GraphChi PR Workload

performance compared to the one used for running the other Cassandra workloads. Still regarding Feedzai's workload, CMS shows shorter GC pauses for lower percentiles but shows the worst results in higher percentiles (25% worse than G1 and 47% worse than NG2C). G1 shows more stable GC pause times (when compared to CMS) as it does not lead to long pauses in higher percentiles; NG2C shows GC pause times very similar to CMS in lower percentiles, and it shows shorter GC pause times for higher percentiles as well.

The other Cassandra workloads (WI, WR, and RI, Figures 6.10 to 6.12) differ only in the percentage of read and writes. From the GC perspective, more writes means that more objects are kept in memory (which results in more object copies and therefore longer GC pauses). This obviously applies to Cassandra because it buffers writes in memory. This is clearly observable by comparing the GC pauses across the three workloads (WI, WR, and RI) for CMS and G1. RI workload shows shorter GC pauses than WR and WI, while WR shows shorter pauses than WI but longer than RI. According to our results, CMS is more sensitive to writes (than the other two collectors) as it has a steep increase in the GC pause time as we move towards write intensive workloads. G1 has a more moderate increase in GC pause time in more intensive workloads.

Regarding NG2C, it produces a different behavior as it shows shorter GC pauses for lower percentiles in WI, and longer pauses for WR in higher percentiles. One factor contributes for this difference (between NG2C, and G1 and CMS): we mainly tuned our code changes in Cassandra for the WI workload (worst case, where more objects are created). This means that the read path is not as optimized as the write path. Therefore, in write intensive workloads, NG2C is more optimized than in read intensive workloads. This is also observable by measuring the difference between the GC pause times in higher percentiles; as we move towards write intensive workloads, the difference between NG2C and other GCs increases.

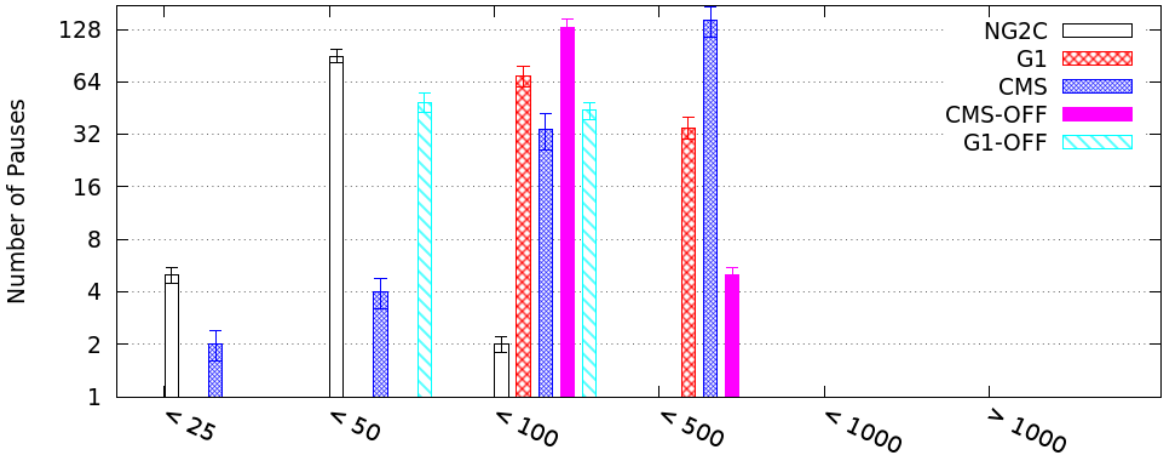

Figure 6.17: Application Pauses Per Duration Interval (ms) for Cassandra WI Workload

We also have results for Cassandra with the off-heap memory enabled for CMS and G1 (Figures 6.10 to 6.12). Cassandra uses off-heap memory to store values while the keys remain in the managed heap. Using off-heap reduces GC pause times by up to 50% in the WI workload (versus 93.8% using NG2C), around 20% in the WR workload (versus 39% using NG2C), and shows no improvement for the RI workload (versus 61% using NG2C). In sum, using NG2C is more effective to reduce GC pause times than using off-heap memory mainly because Cassandra needs to keep header objects in the memory managed heap to describe the contents stored in off-heap. In the case of Cassandra (Key-Value store), keys are stored in the managed heap and therefore contribute for long application pauses. NG2C is able to move all Key-Value pairs into a specific dynamic generation (thus avoiding pause times).

The remaining workloads (Lucene, PR, and CC) are all write intensive (Figures 6.14 to 6.16). CMS shows very high GC pause times compared to the other two GCs. G1 shows a more moderate increase in GC pause times, when compared to CMS, but is still worse than NG2C. In sum, NG2C clearly improves the worst observable GC pause times by: 85% (CMS) and 38% (G1) in Lucene, 97% (CMS) and 84% (G1) in PR, and 97% (CMS) and 82% (G1) for CC.

Figures 6.17 to 6.23 present the average and standard deviation for the number of pauses in different duration intervals. Results show that: i) NG2C does not increase the number of pauses, and ii) it shows more pauses with shorter duration. CMS presents the worst results by having the most amount of pauses in longer pause intervals.

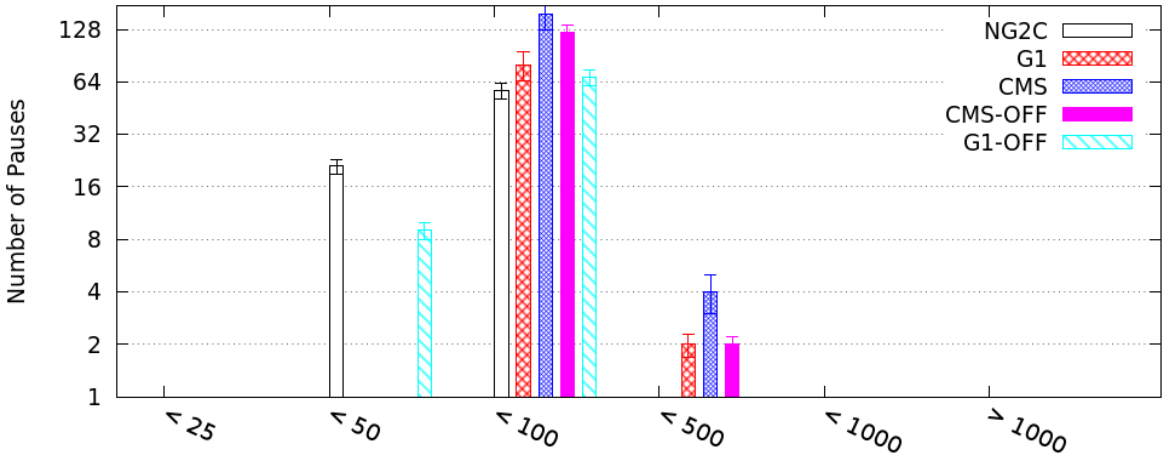

Figure 6.18: Application Pauses Per Duration Interval (ms) for Cassandra RW Workload

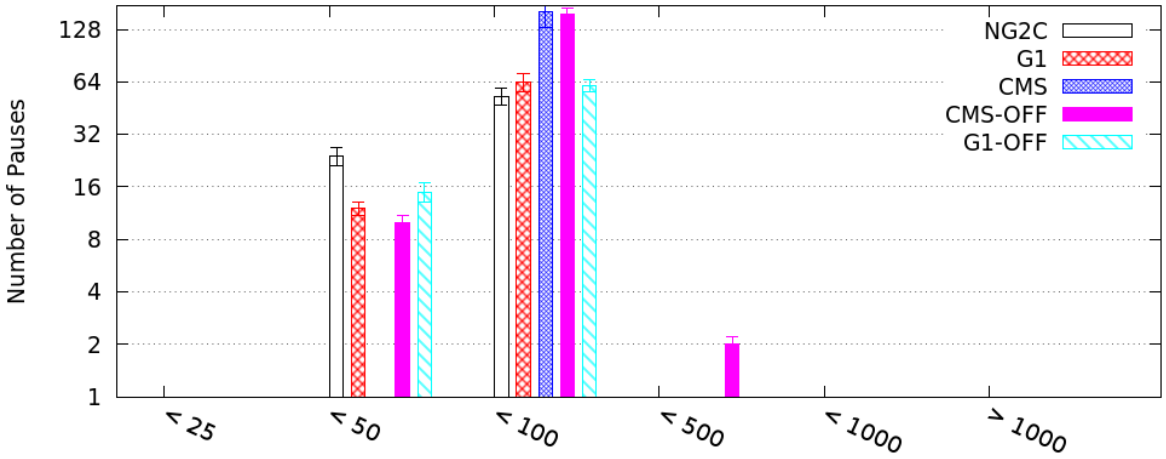

Figure 6.19: Application Pauses Per Duration Interval (ms) for Cassandra RI Workload

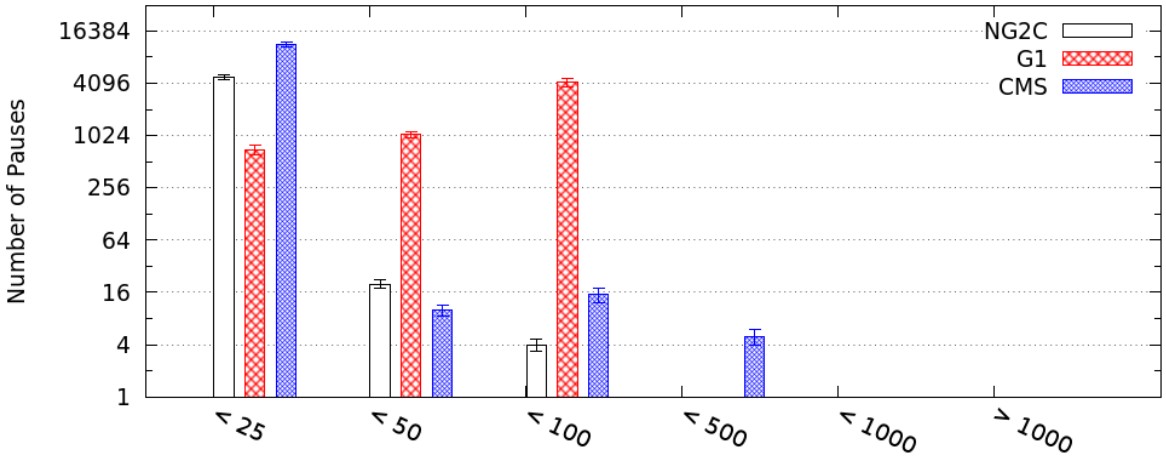

Figure 6.20: Application Pauses Per Duration Interval (ms) for Cassandra Feedzai Workload

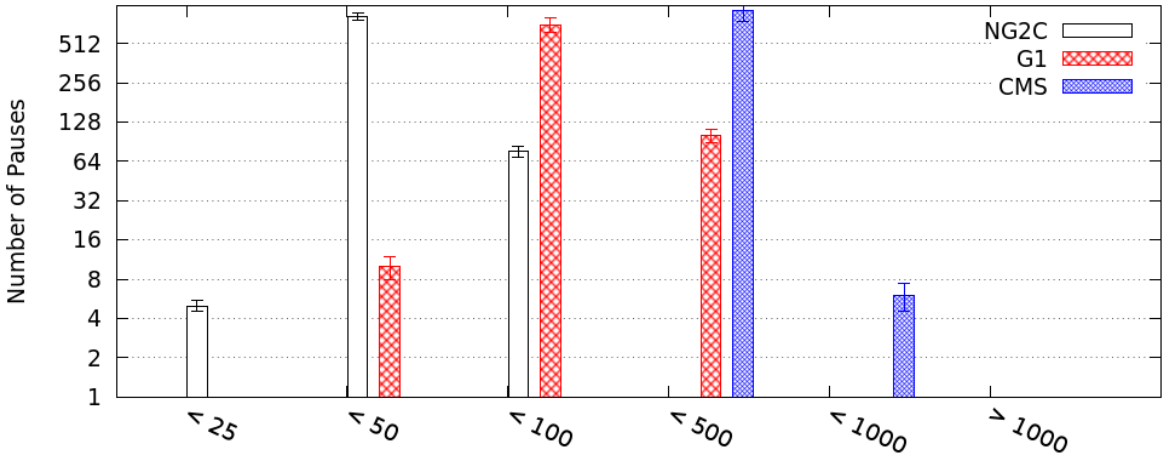

Figure 6.21: Application Pauses Per Duration Interval (ms) for Lucene Workload

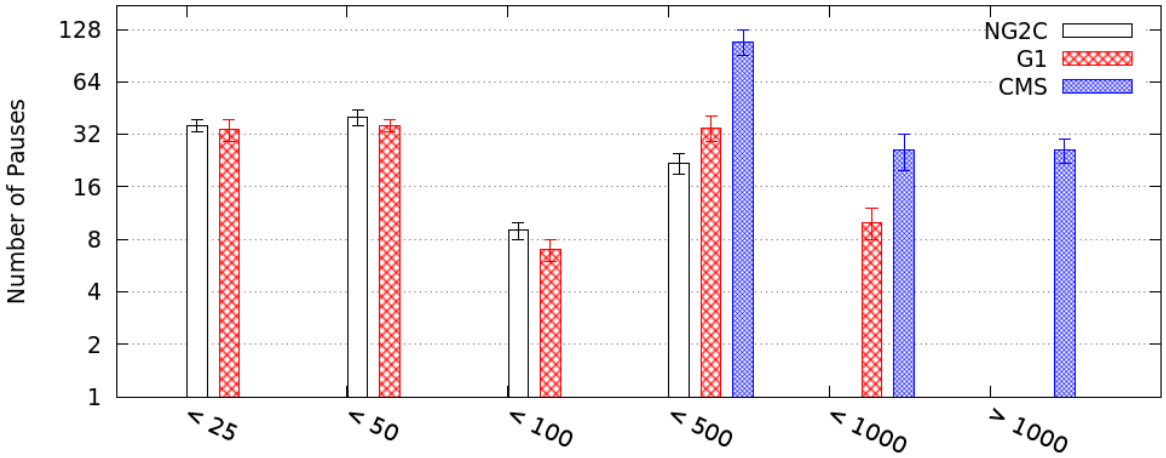

Figure 6.22: Application Pauses Per Duration Interval (ms) for GraphChi CC Workload

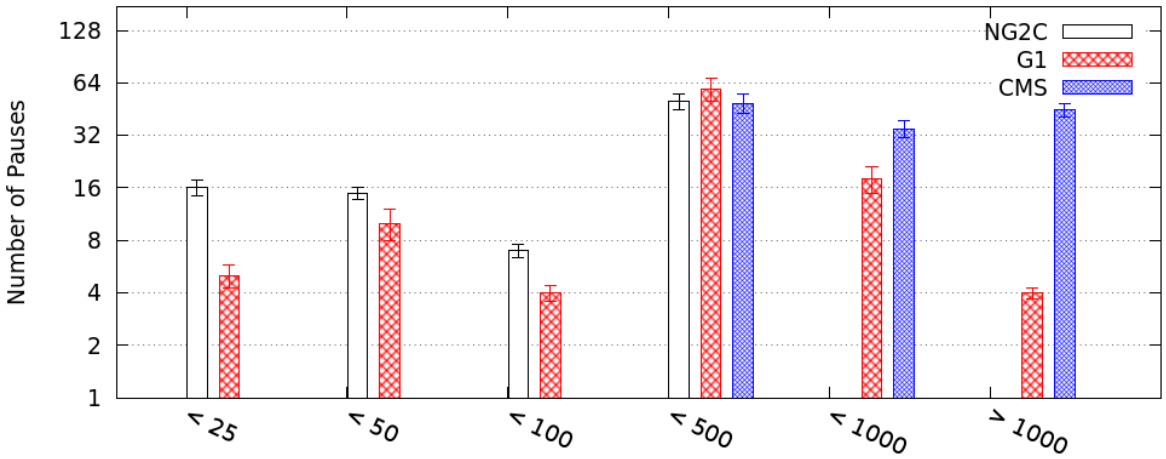

Figure 6.23: Application Pauses Per Duration Interval (ms) for GraphChi PR Workload

6.3.4 Object Copy and Remembered Set Update

We now look into how much time is spent: i) copying objects within the heap, and ii) updating remembered set entries, upon a collection. Note that the remembered set updates is an impor-

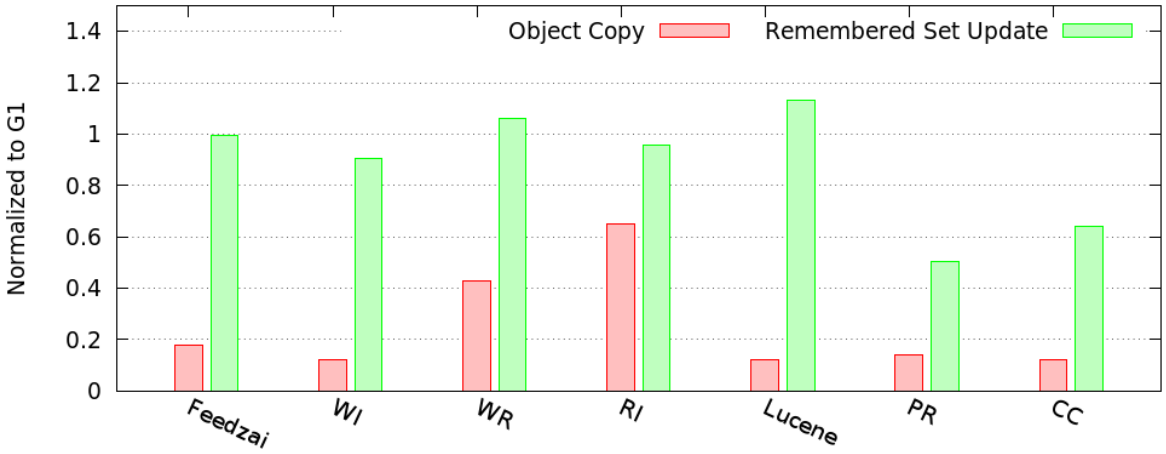

Figure 6.24: NG2C Object Copy and Remembered Set Update, Normalized to G1

tant metric since pretenuring can lead to high number of remembered set updates because of the potential increase in the number of references coming from older to younger spaces [67]. We only show results for G1 and NG2C, given that CMS and C4 do not provide such logging information. However, both metrics are similar for different generational collectors because they mostly depend on: i) the mutator allocation speed (dictates how fast minor collections are triggered and how many objects are promoted), and ii) the available hardware memory bandwidth. Both these factors are kept constant across GCs (G1 and NG2C).

Figure 6.24 presents results for total object copying time and remembered set update time during each workload. All results are normalized to G1. Results show that NG2C reduces objects copying between 30.6% and 89.2%. Note that, in G1, we can not differentiate between object promotion and object compaction since the collector collects both young and old regions at the same time (during mixed collections).

NG2C also has a positive impact for the remembered set update work. This means that, in NG2C, there is not an increase in the number of inter-generational references pointing to the *Young* generation. This is possible because objects referenced by pretenured objects are most likely to be pretenured as well. NG2C even reduces the amount of remembered set update work for most workloads since it reduces the amount of premature promotion in G1 (objects with short life times that were allocated right before a minor collection and were prematurely promoted). This also means that NG2C puts less pressure on the write barrier (compared to G1) used to update the remembered set.

	Max Mem Usage			Throughput		
	CMS	G1	C4	CMS/OFF	G1/OFF	C4
Feedzai	.92	1.00	-	-	-	-
WI	.96	1.01	1.73	1.07/1.08	.99/1.01	.70
WR	.80	1.00	2.04	.76/.90	.93/0.73	.67
RI	.73	.98	1.94	.86/1.18	.90/0.65	.71
Lucene	.39	.98	-	.59	.87	-
PR	1.44	1.04	-	.80	.96	-
CC	1.43	1.17	-	1.03	.96	-

Table 6.7: Max Memory Usage and Throughput norm. to NG2C (i.e., NG2C value is 1 for all entries)

6.3.5 Memory Usage

In this section, we look into the max memory usage to understand how NG2C relates to other collectors regarding heap requirements (see Table 6.7). Regarding the workloads' max heap size: Feedzai workload has 30 GB, while the other Cassandra workloads (WI, WR, and RI) have 12 GB; each Lucene and GraphChi's workload (PR and CC) have 120 GB.

From Table 6.7 we can conclude that, regarding Cassandra workloads (i.e., Feedzai, WI, WR, and RI) all collectors (excluding C4) have a very similar max memory usage. CMS has a slightly smaller heap (compared to G1 and NG2C) while NG2C has a slightly larger heap (compared to G1 and CMS). This slight increase comes from the fact that dynamic generations are only collected upon a mixed collection, which is only triggered when the heap usage is above a configurable threshold. This can lead to a slight delay in the collection of some objects that are already unreachable. C4 has a considerably higher memory usage since it reserves approximately 75% (12 GB) of the system's memory, when the JVM is launched (this comes from an implementation design decision to reduce memory pressure and allow shorter pause times). We do not show the results for C4 with other workloads because we only have one license (for one physical node).

Lucene max memory utilization is lower for CMS when compared to G1 and NG2C. These larger heap sizes in G1 and NG2C comes mostly from humongous allocations. Using this technique, very large objects are directly allocated in the *Old* generation. It has the clear drawback of delaying the collection of such very large objects. Since CMS does not have such technique (i.e., all objects are allocated in the `Eden`), CMS tries to collect these large objects upon each minor collection, leading to faster collection of such objects, thus achieving lower heap usage. Comparing G1 with NG2C, the heap usage is similar.

Regarding GraphChi (PR and CC), it shows a different memory behavior when compared to Cassandra and Lucene, as it allocates mostly small objects. Most of these small objects

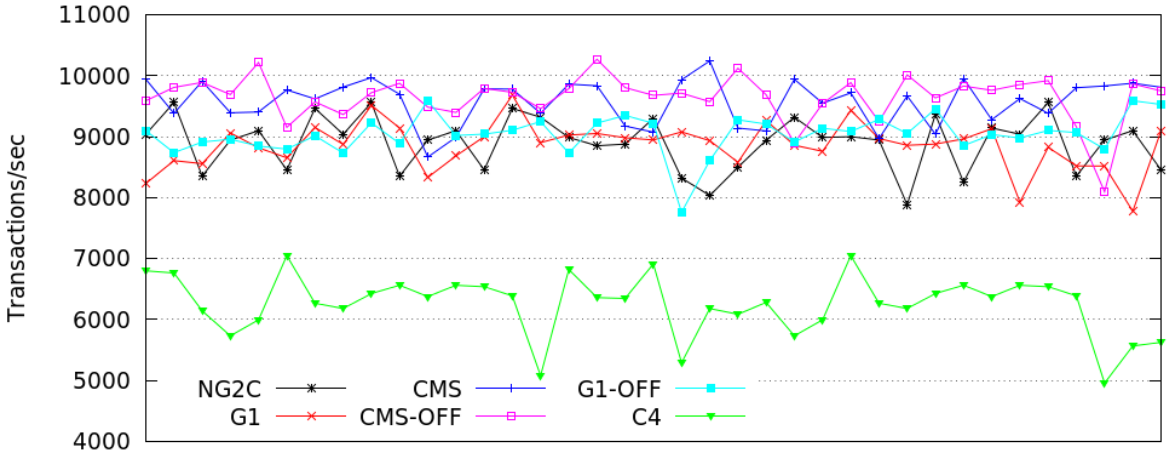

Figure 6.25: Cassandra WI Throughput (transactions/second) - 10 min sample

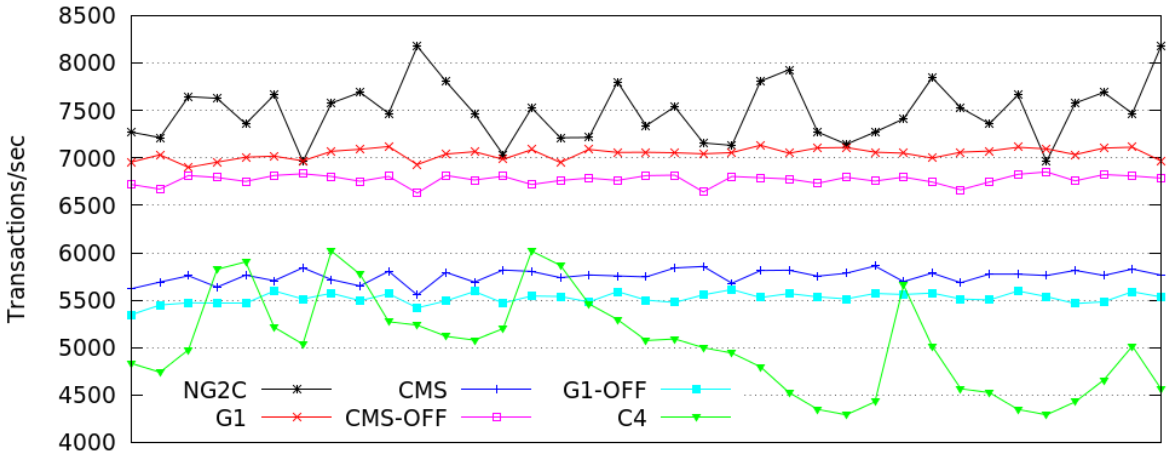

Figure 6.26: Cassandra WR Throughput (transactions/second) - 10 min sample

(mostly data objects representing vertexes and edges) are used in a single iteration, which is long enough for them to be promoted into the *Old* generation (in the case of CMS and G1). Since we set the maximum heap size to 120 GB, the heap fills up until a concurrent marking cycle is triggered. In CMS, the concurrent marking cycle is triggered a bit later compared to G1 and NG2C (thus leading to an increase in the max heap usage). Regarding G1 and NG2C, both present similar max heap values.

6.3.6 Application Throughput

We now discuss the throughput obtained for each GC and workload (except for Feedzai). We do not show the throughput for Feedzai's workload because the benchmark environment (where the Cassandra cluster is used) dynamically adjusts the number of transactions per second according to external factors; e.g., the credit card transaction generator produces different trans-

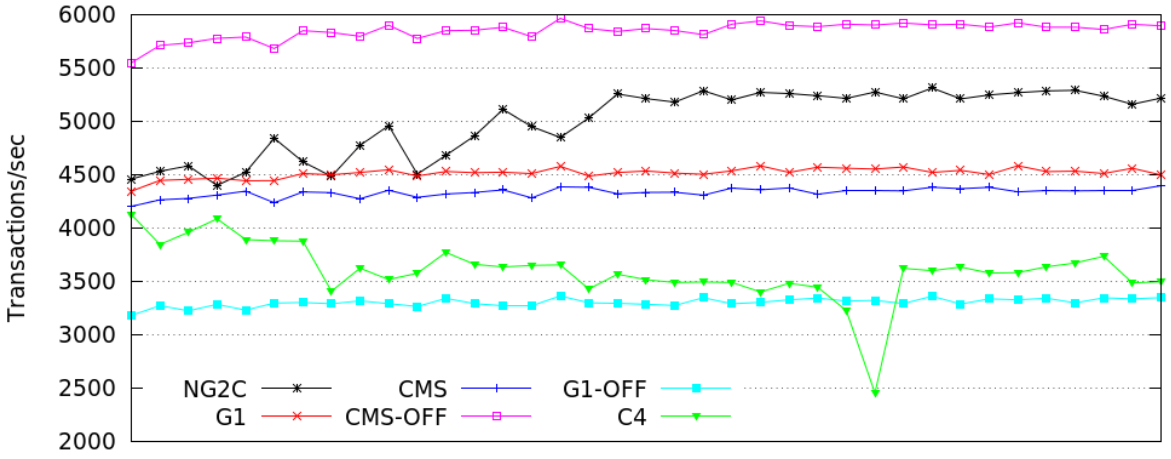

Figure 6.27: Cassandra RI Throughput (transactions/second) - 10 min sample

actions through time, some result on more Cassandra transactions than others, thus making it infeasible to reproduce the same workload multiple times. The throughput for all remaining workloads is presented in Table 6.7. Throughput for Cassandra using off-heap is shown for WI, WR, and RI workloads. All results are normalized to NG2C.

From Table 6.7, we conclude that NG2C outperforms CMS, G1, and C4 (we could only obtain results for Cassandra workloads using C4 because we only have one license) for most workloads. Figures 6.25 to 6.27 show the throughput evolution for Cassandra workloads. NG2C is the solution with overall best throughput across the three workloads. Only the CMS collector using off-heap outperforms NG2C in the read intensive workload (by approximately 18%).

However, as we will see next, CMS shows a steep increase in throughput when configured for throughput (i.e., GC configuration used to maximize application throughput). However, as we ran CMS with a latency oriented configuration in the last experiments, throughput was lower than G1 and NG2C. When comparing G1 with NG2C, NG2C always achieves better throughput. This comes from the fact that there are less and shorter GC pauses stopping the application from progressing (when NG2C is used).

For all previous experiments, we use latency oriented GC configurations, i.e., the configurations we found to enable shorter GC pause times in higher percentiles. This, however, has the drawback of potentially decreasing the throughput. Among the used workloads, the most explicit example of this throughput decrease is Lucene running with CMS, in which a throughput oriented GC configuration, i.e., the configuration we found to enable higher throughput, could increase the throughput by up to 3x (when compared to the throughput achieved with a latency oriented configuration).

To better understand the trade-off between throughput and latency, we ran the Lucene

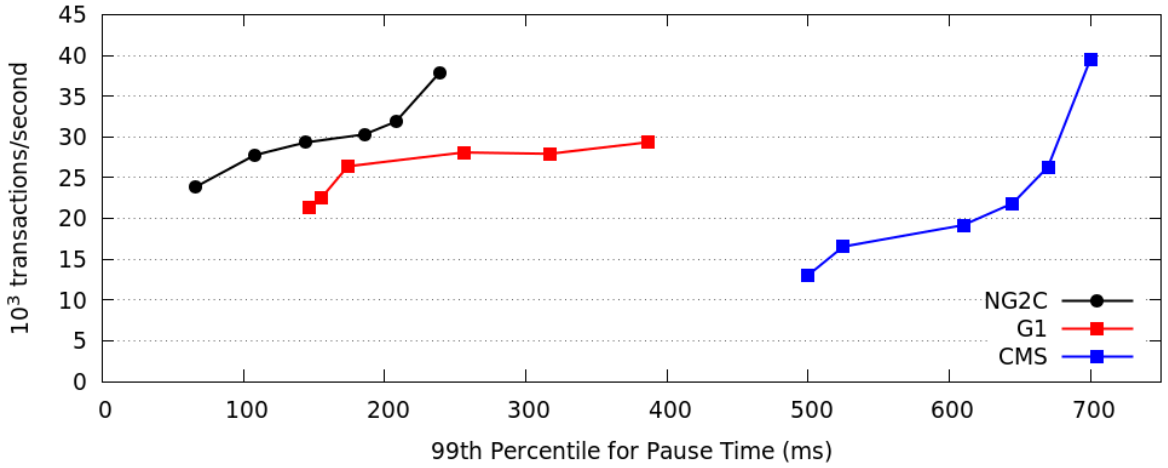

Figure 6.28: Throughput vs Pause Time

workload with 6 different *Young* generation sizes. We found that this parameter alone allows one to achieve good latency (if the size is reduced) or good throughput (if the size is increased). Other GC parameters did not have a relevant effect and therefore we keep them fixed. We start with the configuration used in the previous sections (2 GB). Then, we keep increasing the size of the *Young* generation by 2 GB.

Figure 6.28 shows a plot with the relation between throughput and GC pause time, for each GC, in which each point on each line represents a different *Young* generation size. CMS shows always longer GC pauses independently of the GC configuration. It also shows a steep increase in the throughput, with a small increase in the GC pause time; this shows how easy it is to dramatically reduce throughput when CMS is configured for latency. On the other hand, G1 shows much shorter GC pauses than CMS at the cost of some reduced throughput. Note that moving from latency oriented to throughput oriented configurations has a small impact on throughput, but has a larger negative impact on GC pause time. Finally, NG2C provides the shortest GC pause times with a very small throughput impact. In the most throughput oriented configuration (point on the top of the curve), NG2C is only 5% worse than CMS and the GC pause time is 66% better.

In conclusion, from this section we can extract that: i) CMS can be difficult to configure for short GC pause time (while keeping an acceptable throughput); ii) G1 leads to shorter pauses but can damage throughput; iii) NG2C keeps up with the best throughput achieved by CMS, while also reducing the GC pause times by 66% and 39% w.r.t. CMS and G1, respectively.

6.4 POLM2's Evaluation

This section presents the results of several experiments regarding the three most relevant metrics for POLM2: i) the size of JVM snapshots, and the time to create them; ii) application pause times; and iii) application throughput and memory usage.

Thus, for the first set of results, we compare the size and time to create JVM snapshots with *Dumper*, and the widely used JVM tool `jmap`. Then, application pause times obtained with POLM2 are analyzed and compared to the pauses obtained with: i) NG2C with manual code modifications, i.e. with programmers' knowledge of the application and the corresponding source code modifications (to allocate objects with similar life times close to each other), and ii) G1, the default collector for recent OpenJDK versions that uses no profiling information or programmers' help to estimate objects life time, and therefore simply assumes that most objects die young. Note that the use of NG2C with manual code modifications should correspond to the best performance results that can be obtained as the programmer knows the application code and requires objects to be allocated in a way that minimizes pause times. Although we also use C4 [113] in our experiments, pause time results are not shown as there are no significant pause times (the duration of all pauses fall below 15 ms).

We use three relevant platforms (that are used in large-scale environments) to exercise each collector approach: i) Apache Cassandra 2.1.8 [79], a large-scale Key-Value store, ii) Apache Lucene 6.1.0 [90], a high performance text search engine, and iii) GraphChi 0.2.2 [78], a large-scale graph computation engine. These are the same platforms used to evaluate NG2C and whose workload description was presented in Section 6.1.

The main goal of these evaluation experiments is to show that POLM2: i) greatly reduces application pause times when compared to current industrial collectors (such as G1); ii) does not have a negative effect neither on throughput nor on memory utilization; and iii) replaces programmer's knowledge by automatic profiling, thus achieving equivalent performance or even outperforming NG2C.

6.4.1 Evaluation Environment

The evaluation was performed using a server equipped with an Intel Xeon E5505, with 16 GB of RAM. The server runs Linux 3.13. Each experiment runs in complete isolation for at least 5 times (i.e., to be able to identify outliers). All workloads run for 30 minutes each. When running each experiment, the first five minutes of execution are ignored; this ensures minimal interference from JVM loading, JIT compilation, etc.

Workload	# Instr. Alloc. Sites	# Used Generations	# Conflicts Encountered
Cassandra-WI	11/11	4/N	2/2
Cassandra-RW	11/11	4/N	2/2
Cassandra-RI	10/11	4/N	3/2
Lucene	2/8	2/2	2/0
GraphChi-CC	9/9	2/2	1/0
GraphChi-PR	9/9	2/2	1/0

Table 6.8: Application Profiling Metrics for POLM2/NG2C

Fixed heap and young generation sizes are always enforced (12 GB and 2 GB, respectively). We found that these sizes are enough to hold the working set in memory and to avoid premature en masse promotion of objects to older generations. Besides, according to our experience, leaving young generation or total heap sizes unlimited leads to non-optimal pause times since the JVM will always try to limit/reduce memory utilization, eventually leading to extra GC effort that results in longer pause times.

6.4.2 Application Profiling

This section shows the results obtained during the profiling phase (while the next sections present results obtained during the production phase). Please refer back to Section 4.4.5 where we explain both phases.

Because each workload stabilizes very fast after the JVM loading, we found that profiling each workload for only five minutes is sufficient (the first minute of execution is always ignored to avoid load-time noise; hence, the profiling phase lasts for six minutes per workload). If other workloads take more time to stabilize, the duration of the profiling phase might increase.

Profiling Allocation Sites

For each application and workload, a number of allocation sites are identified as candidates for instrumentation. Each allocation site can be selected for a different generation, according to the estimated life time of objects allocated through that particular allocation site. Finally, encountered conflicts are also solved, resulting in additional code changes (see Section 4.4.3 for more details).

Table 6.8 shows the above mentioned applications profiling metrics for each workload, for both POLM2 and NG2C. Regarding the number of instrumented allocation sites, both POLM2 and NG2C are very similar. For Cassandra-RI and Lucene, POLM2 did not consider as many allocation sites as NG2C which, as discussed in the next sections, has a small positive impact on application latency. Regarding the number of generations used, the only difference is on

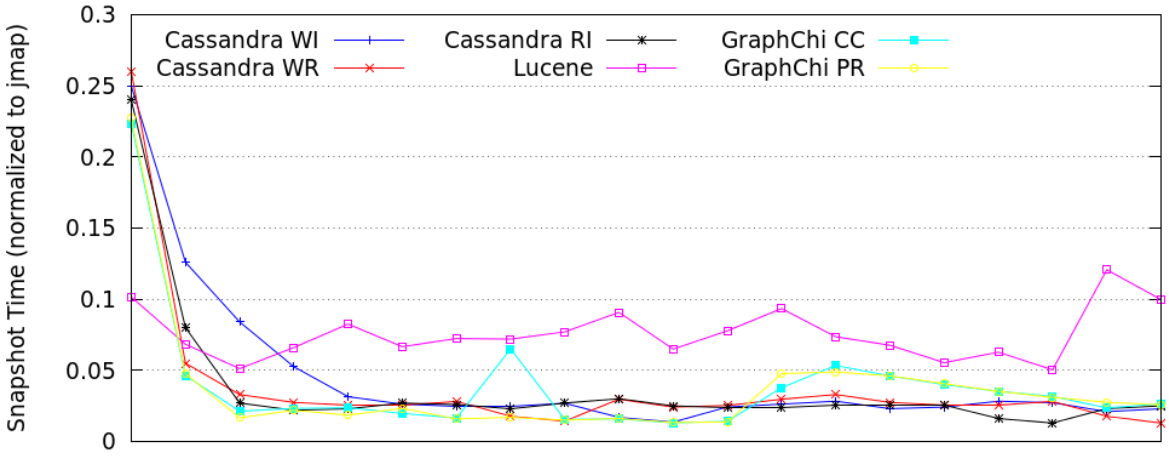

Figure 6.29: Memory Snapshot Time using *Dumper* normalized to `jmap`

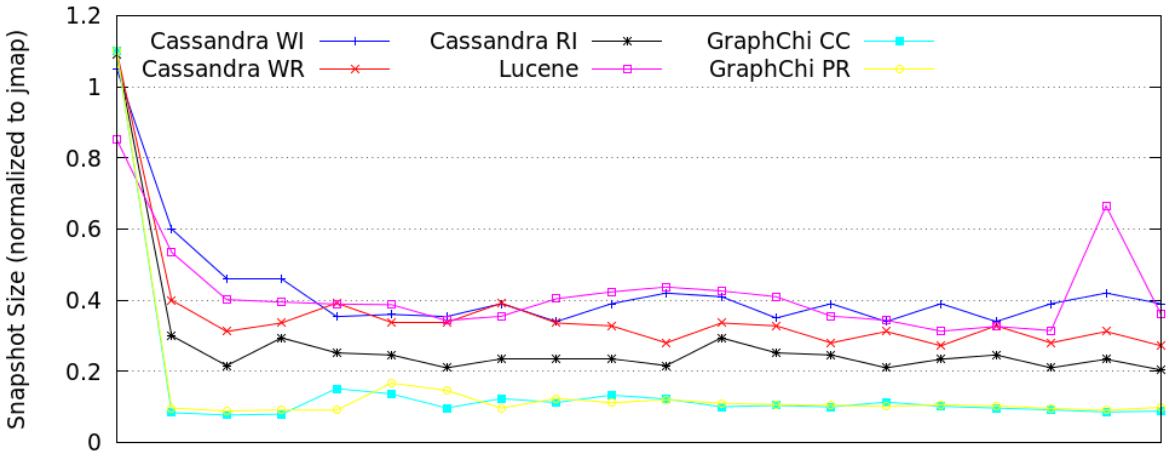

Figure 6.30: Memory Snapshot Size using *Dumper* normalized to `jmap`

how Cassandra workloads are handled; POLM2 uses only four generations while NG2C uses an many number of generations (in fact, NG2C creates one generation each time a `Memtable` is flushed; the new generation is used to accommodate the new `Memtable`). As can be seen in the next sections, this difference has no performance impact (neither positive nor negative).

Finally, POLM2 is able to detect conflicts in Lucene, GraphChi-CC, and GraphChi-PR, that were not correctly identified in NG2C (a conflict found when a code location can produce objects with different life times). As discussed in the next section, this leads to some performance penalties for NG2C.

JVM Memory Snapshots

As discussed in Section 4.4.2, the *Dumper* component uses two optimizations to reduce the overhead associated to taking snapshots: i) it avoids pages with no reachable objects, and ii) it

avoids pages that were not modified since the last snapshot. Figures 6.29 and 6.30 show the results for the first 20 memory snapshots obtained for each workload. Both plots compare the performance of the *Dumper* with `jmap`, a widely used tool that takes JVM heap dumps (in this case, only live objects are dumped with `jmap`). Results are normalized to `jmap`.

From both figures, it is clear that, when compared to `jmap`, POLM2: i) is more efficient creating memory snapshots as it reduces the time necessary to take a snapshot by more than 90% for all workloads, and ii) reduces the size of the snapshots by approximately 60% for all workloads. By enabling faster and cheaper memory snapshots, POLM2 reduces the time applications are stopped to let the profiler snapshot the memory, thus reducing the profiler impact on application performance. As an example, GraphChi (both PR and CC) snapshotted using `jmap` results in a 3.8 GB heap dump (on average), taking 22 minutes to create the snapshot (on average). Using the *Dumper*, the size of the snapshot is reduced to approximately 700 MB, taking approximately 32 seconds to create the snapshot.

6.4.3 GC Pause Times

This section shows results for the GC pause times, i.e., the amount of time during which an application is stopped to let the collector work (collect dead objects). The goal of this section is to demonstrate that, with POLM2, the GC pause times are: i) shorter when compared to G1 (which uses no information regarding objects life times), and ii) as good as those obtained with NG2C (which requires manual code changes). Note that these NG2C hints used by POLM2 are produced during the production phase, i.e., we are taking advantage of the application allocation behavior recorded during the profiling phase.

Pause Time Percentiles

Figures 6.31 to 6.36 present the results for application pauses times across all workloads for POLM2, NG2C, and G1. Pauses are presented in milliseconds and are organized by percentiles (from percentile 50th to percentile 99.999th). The worst observable pause is also included.

As can be seen in the figures, POLM2 clearly outperforms G1, the default collector in OpenJDK HotSpot JVM, across all pause time percentiles. For Cassandra, the worst observable pause times are reduced by 55%, 67 %, and 78% (for WI, RW, and RI respectively). The same applies to the other workloads, where POLM2 also outperforms G1: 78%, 80%, and 58% reduction in the worst observable pause time for GraphChi CC, GraphChi PR, and Lucene (respectively).

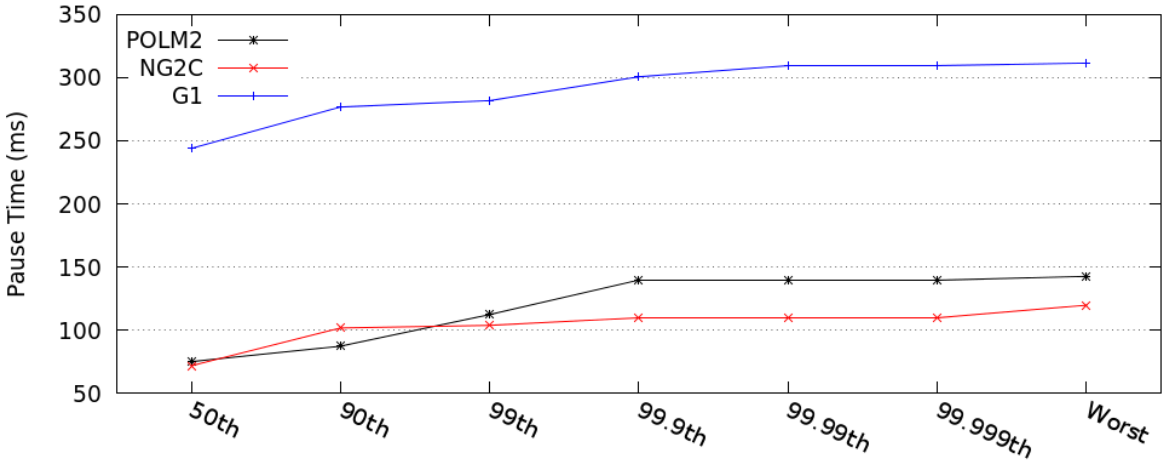

Figure 6.31: Pause Time Percentiles (ms) for Cassandra WI Workload

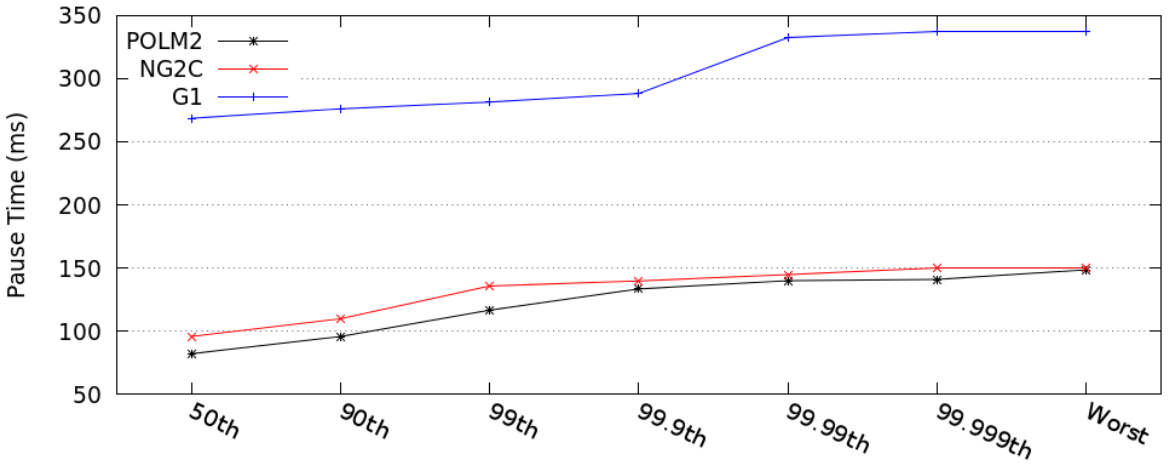

Figure 6.32: Pause Time Percentiles (ms) for Cassandra WR Workload

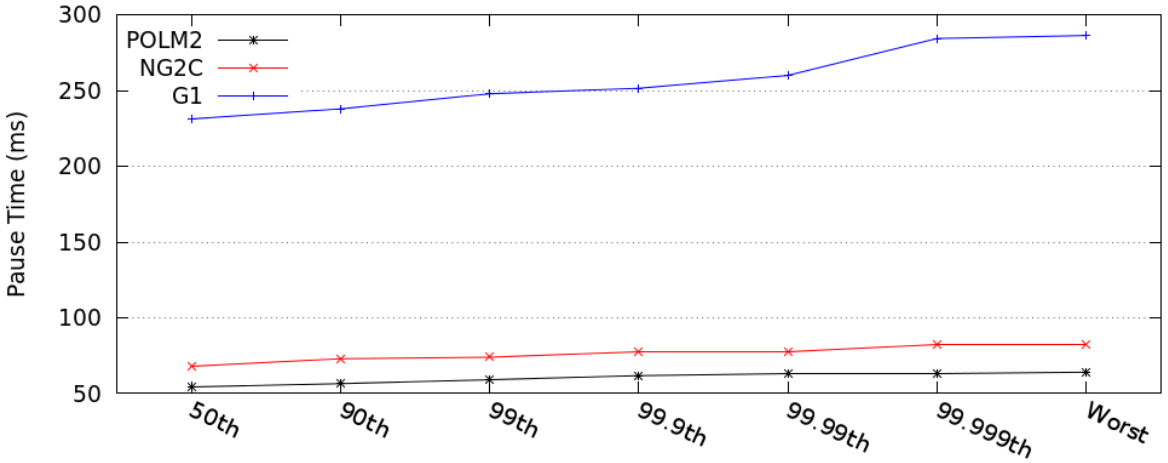

Figure 6.33: Pause Time Percentiles (ms) for Cassandra RI Workload

When comparing POLM2 to NG2C, it is possible to conclude that both solutions achieve similar pause times across all percentiles. However, note that with POLM2, the programmer

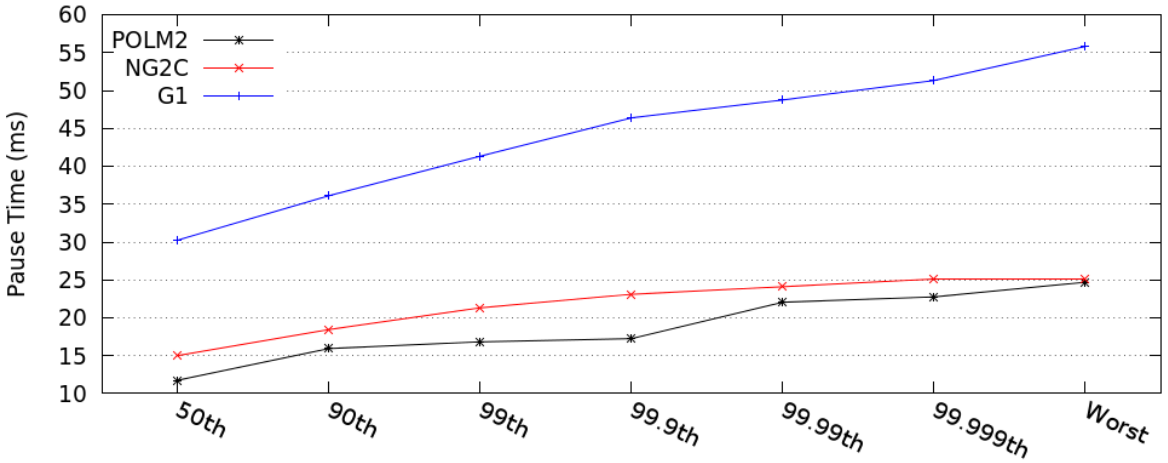

Figure 6.34: Pause Time Percentiles (ms) for Lucene Workload

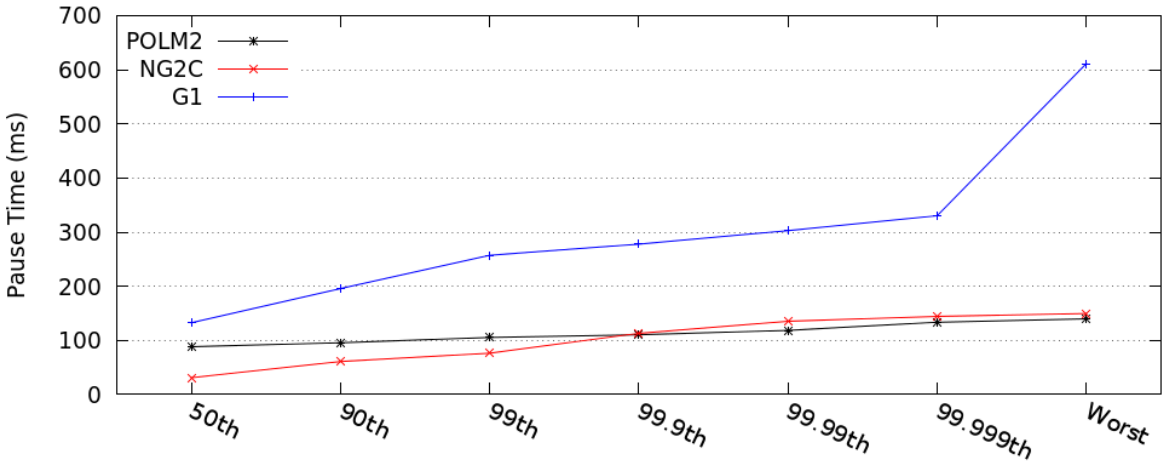

Figure 6.35: Pause Time Percentiles (ms) for GraphChi CC Workload

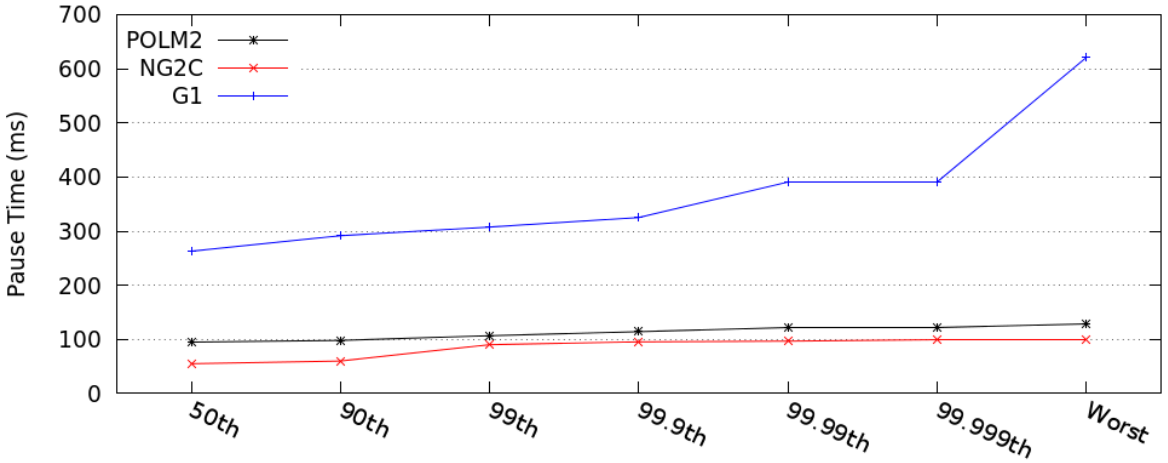

Figure 6.36: Pause Time Percentiles (ms) for GraphChi PR Workload

does not have to change the code; we believe that this is a very important aspect as long as the overall performance of applications is not affected (which is the case). The two workloads

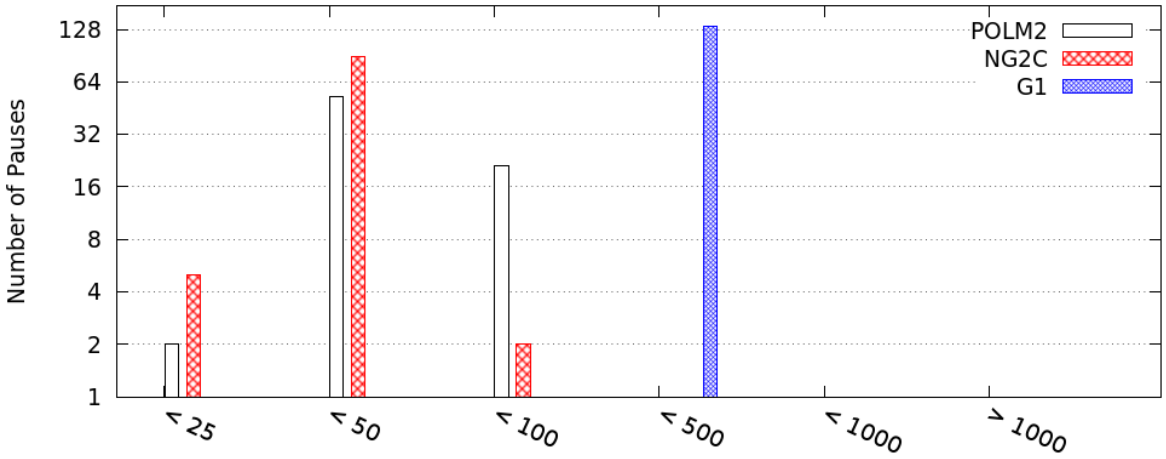

Figure 6.37: Application Pauses Per Duration Interval (ms) for Cassandra WI Workload

that have slightly different results are Cassandra RI, and Lucene. For these workloads, POLM2 is able to slightly outperform NG2C for most percentiles. After analyzing the results, it was possible to determine that the reason behind this difference is related to some misplaced manual code changes.

This an interesting result; it shows that it is very difficult, even for an experienced developer who spends many hours working on the code, to be able to accurately tell: i) which objects live longer than others, and ii) how to set the target generation for each allocation site, taking into consideration that the same allocation site might be used through different allocation paths. This problem is solved by POLM2 using STTrees (as described in Section 4.4.3) that detects these conflicts and properly place calls into NG2C to change the current generation.

In sum, even experienced developers can (and will probably) fail to take into account all the possible allocation paths into a given allocation site. This will result in less optimal object location in the heap, leading to long application pauses. POLM2 solves this problem automatically with no programmer effort.

Pause Time Distribution

The previous section presented application pause times organized by percentiles, which is good for analyzing worst case scenarios but might hide the real pause time distribution. Hence, this section presents the same metric (pause times) but organized in pause time intervals. Figures 6.37 to 6.42 presents the number application pauses that occur in each pause time interval. Pauses with shorter durations appear in intervals to the left while longer pauses appear in intervals to the right. In other words, the less pauses to the right, the better.

As seen in the pause time percentiles, POLM2 brings significant improvements when com-

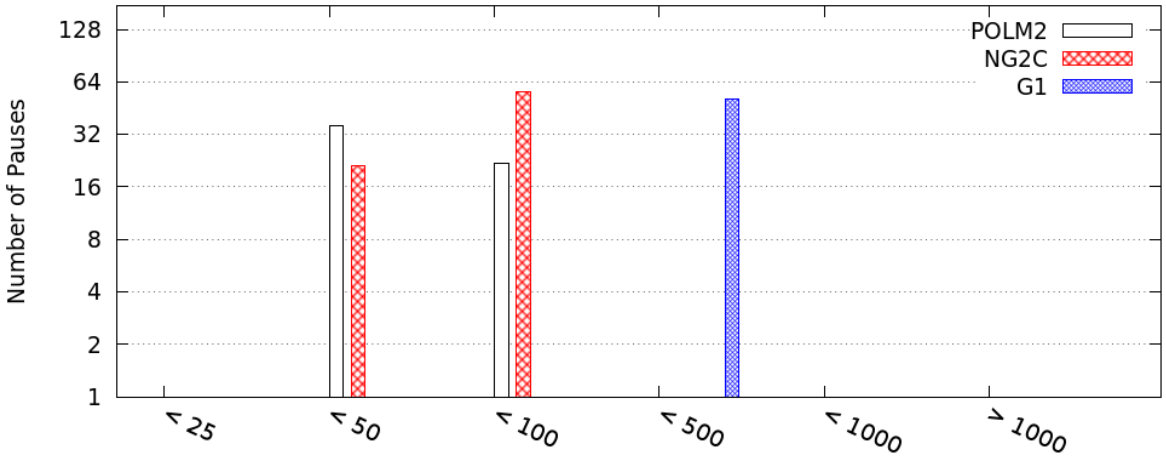

Figure 6.38: Application Pauses Per Duration Interval (ms) for Cassandra RW Workload

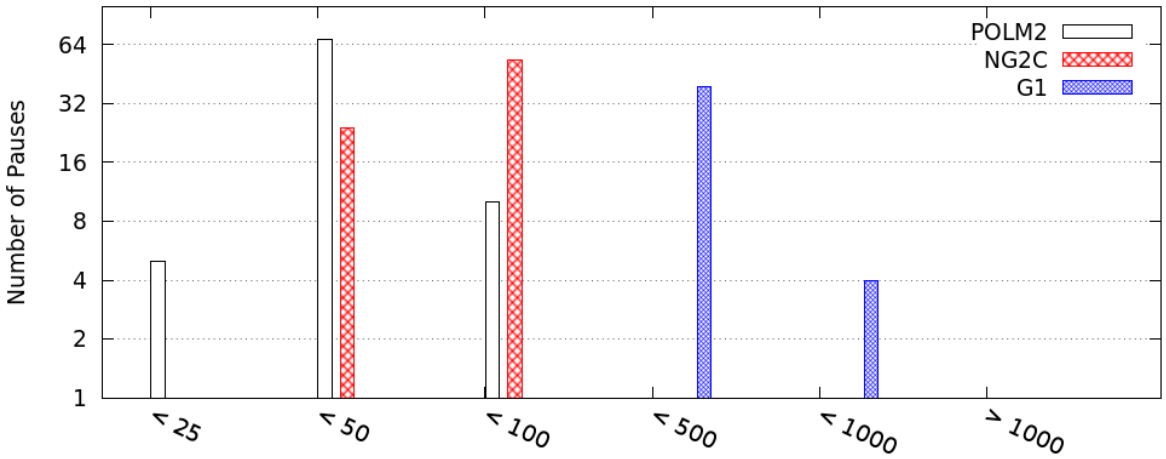

Figure 6.39: Application Pauses Per Duration Interval (ms) for Cassandra RI Workload

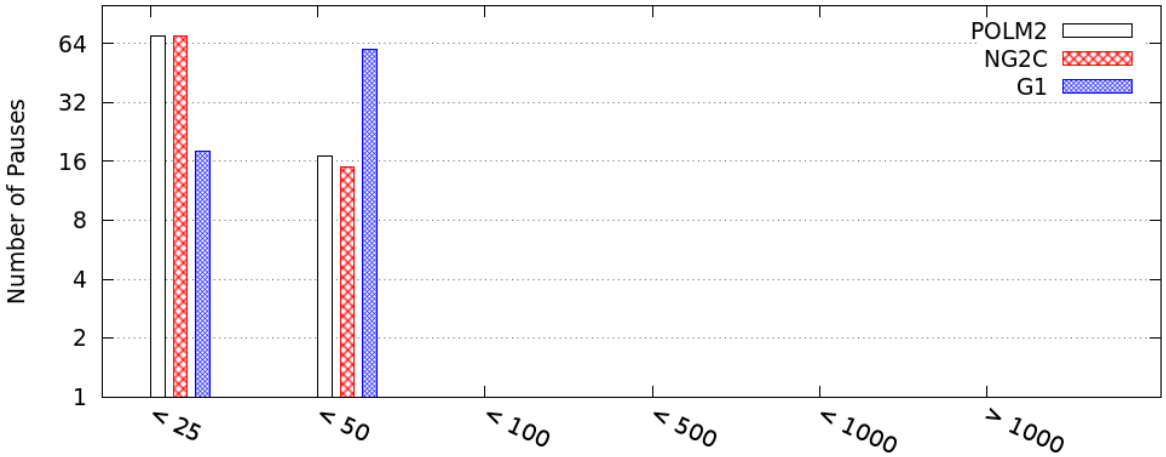

Figure 6.40: Application Pauses Per Duration Interval (ms) for Lucene Workload

pared to G1 as it leads to less application pauses in longer pause intervals. This is true for all workloads. This is an important result because it shows that POLM2 leads to reduced appli-

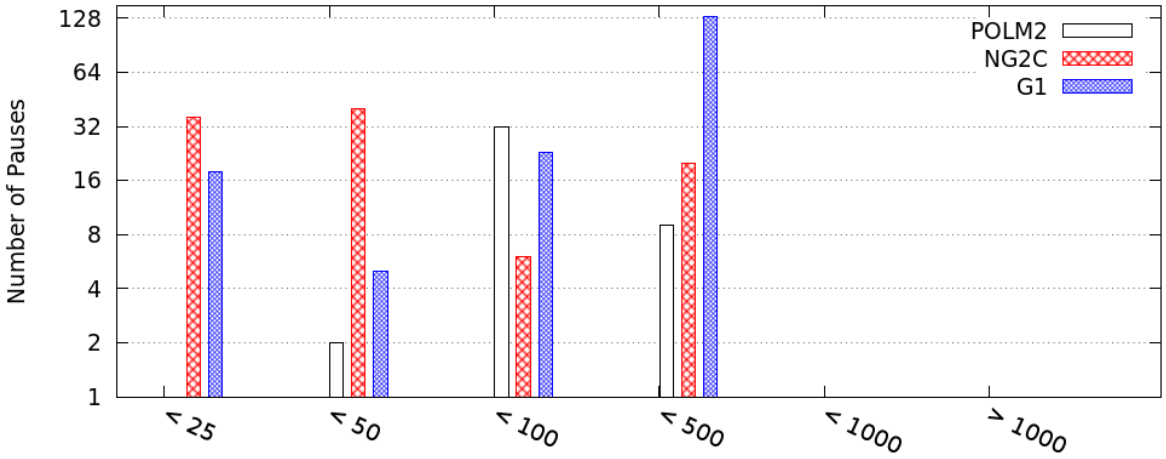

Figure 6.41: Application Pauses Per Duration Interval (ms) for GraphChi CC Workload

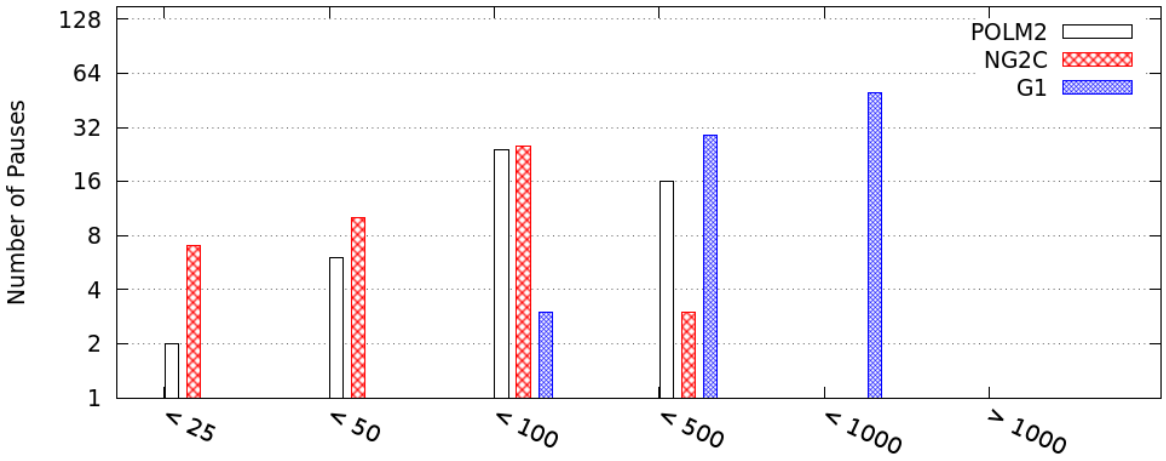

Figure 6.42: Application Pauses Per Duration Interval (ms) for GraphChi PR Workload

cation pauses not only in worst case scenarios (higher percentiles) but also in shorter pause intervals. Thus, it is possible to conclude that POLM2 automatically reduces the duration of all pauses and not only the longer ones.

When comparing POLM2 with NG2C, the same conclusion taken in the previous section holds: POLM2 outperforms NG2C for Cassandra RI and Lucene because of the difficulty in correctly applying NG2C calls/annotations; this can be extremely tricky because of multiple allocation paths for the same allocation site.

6.4.4 Throughput and Memory Usage

This section shows results on application throughput and max memory usage for G1, NG2C, and POLM2. In addition, throughput results for C4 are also shown for Cassandra workloads. Results for max memory usage are not presented for C4 since this collector pre-reserves all

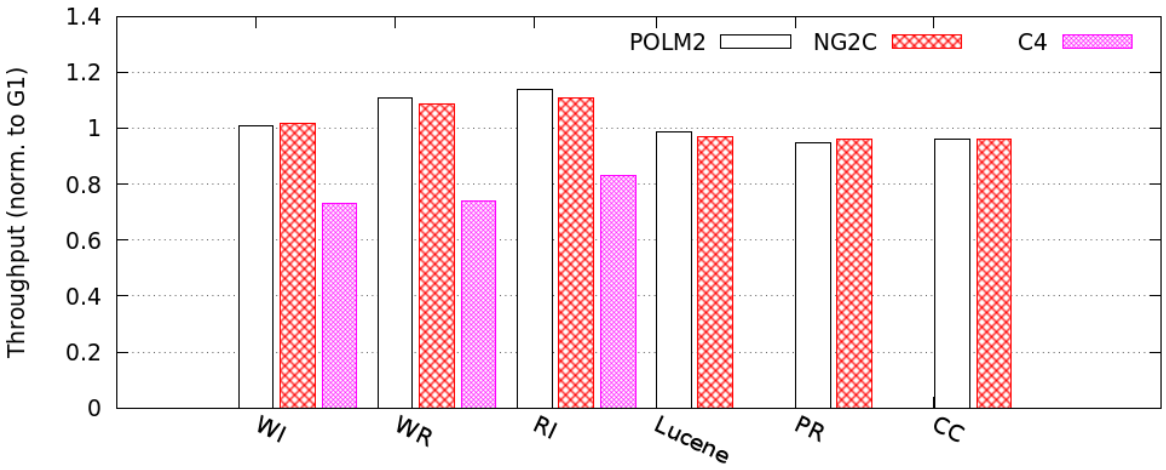

Figure 6.43: Application Throughput normalized to G1

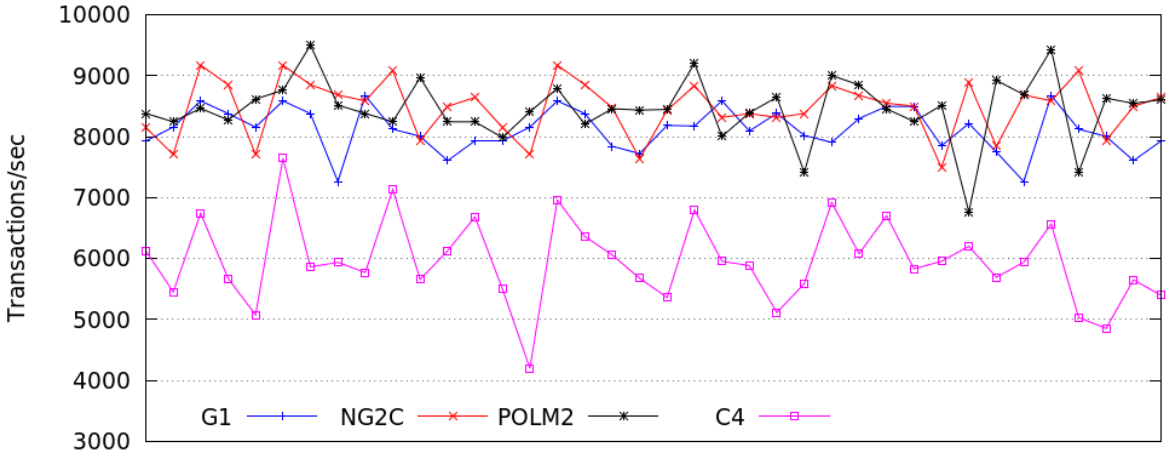

Figure 6.44: Cassandra WI Throughput (transactions/second) - 10 min sample

the available memory at launch time, meaning that, in practice, its memory usage is equal to the max available memory at all time. The goal of this section is to demonstrate that POLM2: i) does not inflict a negative throughput impact, and ii) does not negatively impacts the max memory usage.

Figure 6.43 shows the application throughput for NG2C, C4, and POLM2. Results are normalized to G1 (meaning that if G1 was also plotted, it would have one in all bars). Results show that, not only POLM2 does not negatively impacts throughput, but even improves it. Comparing POLM2 to G1, POLM2 is able to improve throughput by 1%, 11%, and 18% for Cassandra WI, WR, and RI, respectively. Compared to G1, POLM2 leads to a slight reduction in throughput in Lucene (1% loss), GraphChi PR (5% loss), and GraphChi (4% loss). The throughput achieved with POLM2 and NG2C is very similar, with no relevant positive or negative impact on any workload. Once again, the difference is that with POLM2 there is no extra programmer effort. C4 is the collector with worst performance. This overhead comes from the fact that C4 relies

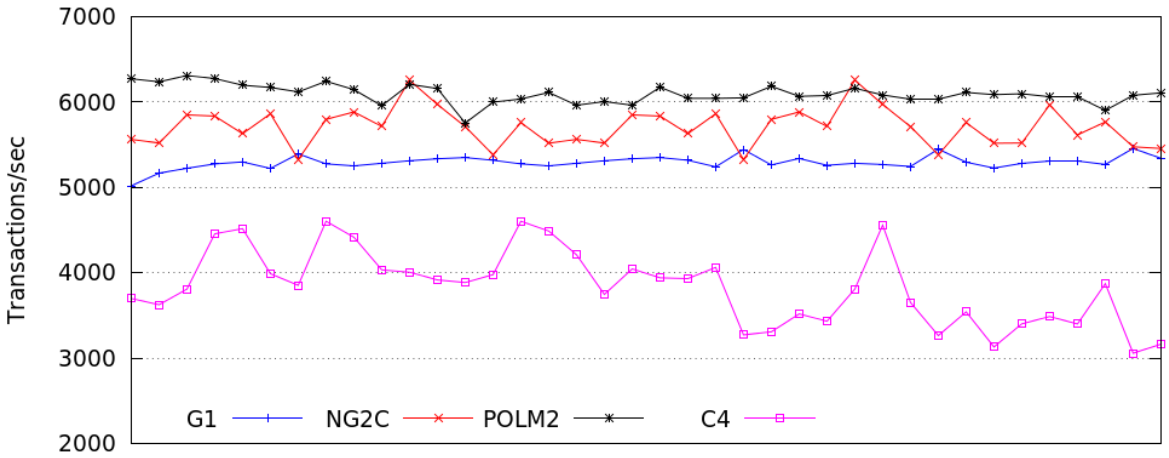

Figure 6.45: Cassandra WR Throughput (transactions/second) - 10 min sample

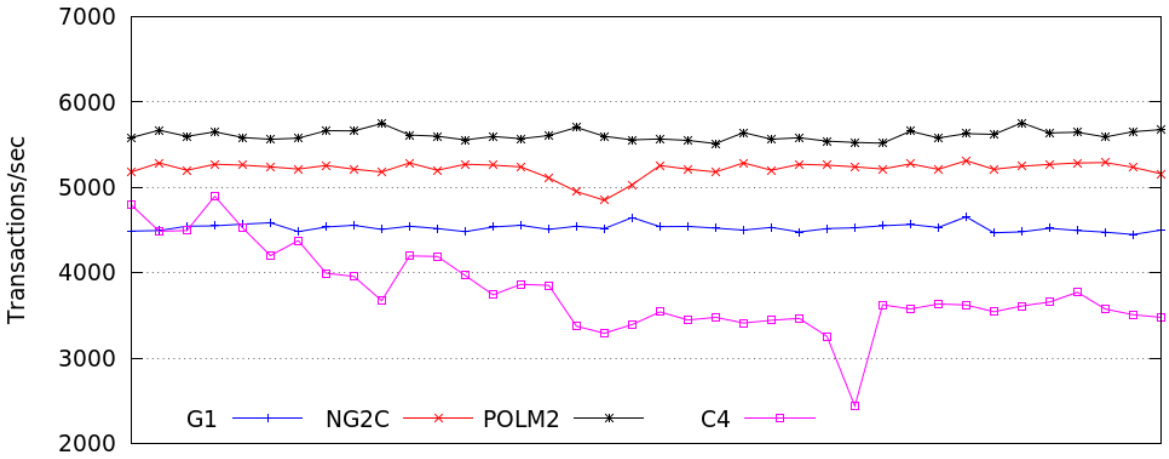

Figure 6.46: Cassandra RI Throughput (transactions/second) - 10 min sample

on several techniques such as a combination of read and write barriers to provide near to zero pause times.

Figures 6.44 to 6.46 show more detailed throughput values for all three Cassandra workloads (WI, WR, and RI). Each plot presents a 10 minute sample containing the number of executed transactions per second. The main conclusion to take from these plots is that the throughput is approximately the same for each approach (G1, NG2C, and POLM2). This means that, for Cassandra, POLM2 does not present any throughput limitation; therefore, it is a better solution compared to NG2C (that requires developer effort), and it is a better solution than G1 since is reduces application pauses. As pointed previously, C4 is the collector with worst performance.

Finally, Figure 6.47 presents the results for max memory usage across all workloads. Again, results are normalized to G1 (meaning that it would have one in all columns). For this particular metric, G1, NG2C, and POLM2 lead to very similar memory usages, meaning that the memory

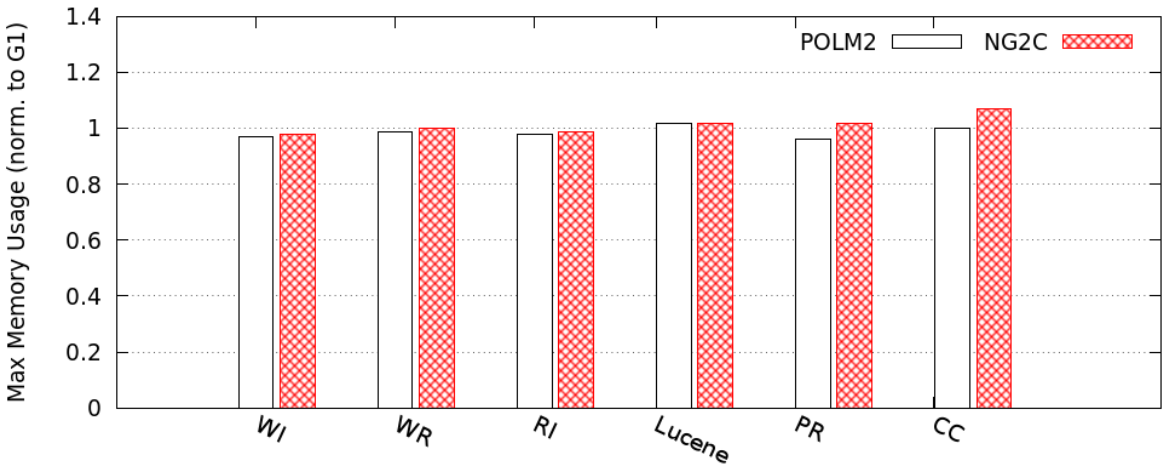

Figure 6.47: Application Max Memory Usage normalized to G1

needed to run each workload is not increased by any of the solutions. This is an interesting result because it shows that it is possible to perform a life time-aware memory management without increasing the application memory footprint. This also means that using multiple generations does not increase external memory fragmentation. C4 results are not shown as it pre-reserves all available memory at launch time. If plotted, results for C4 would be close to 2 for Cassandra benchmarks.

6.5 ROLP's Evaluation

This section provides an exhaustive performance evaluation of ROLP (see Section 4.5). The goal of this evaluation is twofold. First, we need to analyze the performance overhead introduced by ROLP's profiling code. To do that, we take advantage of the DaCapo [13] benchmark suite, which we use to exercise the JVM with different workloads. Using these workloads, we study and break down the performance overheads introduced by ROLP. Second, we measure the pause time improvements of NG2C using ROLP's profiling information. Remember that NG2C uses hand-made annotations to estimate the life time of objects. When using ROLP, we are only relying on runtime information and therefore, we need to measure how well ROLP can replace the programmer effort to estimate objects' life time.

To evaluate the effect of ROLP on application pause times, we use three relevant platforms (that are used in large-scale environments) to exercise each collector approach: i) Apache Cassandra 2.1.8 [79], a large-scale Key-Value store, ii) Apache Lucene 6.1.0 [90], a high performance text search engine, and iii) GraphChi 0.2.2 [78], a large-scale graph computation engine. These are the same platforms used to evaluate NG2C and whose workload descrip-

tion was presented in Section 6.1.

Three systems/collectors are used for this evaluation: i) G1 (see Section 3.2.2, the current default collector in OpenJDK HotSpot JVM; ii) NG2C pretenuring collector (presented in Section 4.3; ii) POLM2, offline profiler, integrated with NG2C, which requires a profiling phase to prepare object life time estimates for each allocation site (presented in Section 4.4); and iii) ROLP, the runtime object life time profiler, integrated with NG2C (presented in Section 4.5).

6.5.1 Evaluation Environment

The evaluation was performed using a server equipped with an Intel Xeon E5505, with 16 GB of RAM. The server runs Linux 4.13. Each experiment runs in complete isolation for 5 times (enough to be able to detect outliers). All workloads run for 30 minutes each. When running each experiment, the first five minutes of execution are discarded to ensure minimal interference from JVM loading, JIT compilation, etc. We also ran experiments such as Cassandra in a cluster environment but, for the purposes of this evaluation, there is no difference between exercising a single Cassandra instance or to use a cluster of Cassandra instances and then look at the GC behavior for each one.

Still regarding the large-scale workloads (Cassandra, Lucene, and GraphChi), heap sizes are always fixed. The maximum heap size is set to 12 GB while the young generation size is set to 2 GB. According to our experience, these values are enough to hold the workings set in memory, and to avoid premature massive promotion of objects to older generations (in the case of G1). We experimented with other heap configurations, which lead to the same conclusions presented throughout this section. Regarding the heap sizes for DaCapo benchmarks, we use the minimum heap size that allows maximum throughput. The used heap sizes are presented in Table 6.9.

6.5.2 Profiling Performance Overhead

This section presents ROLP's overhead in the DaCapo benchmark suite. Note that DaCapo benchmarks are CPU and memory intensive, representing the worst-case scenario for our profiling code. We devised two experiments: i) run each DaCapo benchmark with different levels of profiling to measure the impact of each type of profiling code in the benchmark's performance; and ii) simulate what would be the overhead of the conflict resolution algorithm proposed in Section 4.5.5 and how long it would take in the worst-case scenario.

Figure 6.48 presents the average execution time of each benchmark normalized to G1 (our baseline). Values above one means it took longer than G1 took to execute. For each bench-

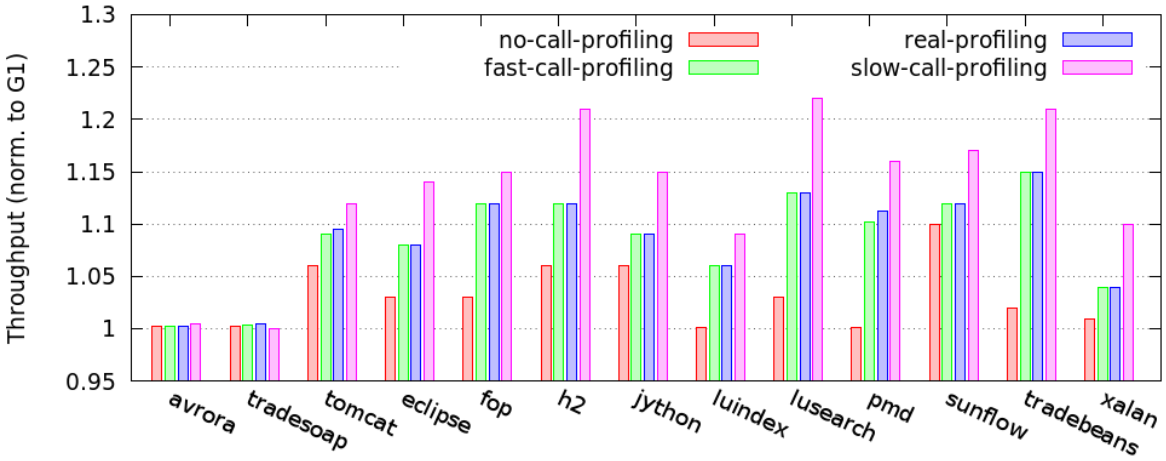

Figure 6.48: DaCapo Benchmark Execution Time Normalized to G1

Benchmark	Heap Size	PMC	PAS	# Conflicts	Conflict Overhead	Conflict Duration
avrora	32 MB	374	69	0	0.04 %	272.00 sec
eclipse	1 GB	1378	329	0	1.20 %	112.00 sec
fop	512 MB	3102	829	0	0.02 %	59.20 sec
h2	1 GB	1416	116	0	1.80 %	320.00 sec
jython	128 MB	11801	741	0	1.20 %	14.40 sec
luindex	256 MB	464	89	0	0.60 %	520.00 sec
lusearch	256 MB	558	127	0	1.80 %	12.00 sec
pmd	256 MB	3157	369	6	1.20 %	87.20 sec
sunflow	128 MB	346	225	0	1.00 %	10.40 sec
tomcat	512 MB	2891	436	4	0.60 %	55.20 sec
tradebeans	512 MB	2145	227	0	1.20 %	91.20 sec
tradesoap	512 MB	5815	254	3	0.60 %	100.00 sec
xalan	64 MB	2037	406	0	1.80 %	6.40 sec

Table 6.9: DaCapo Benchmarks Profiling and Worst-Case Conflict Overhead and Duration

mark, there are four columns (from left to right): i) **no-call-profiling** represents the execution time with no call profiling, i.e., only object allocation is profiled in this experiment and therefore, the execution overhead comes only from the profiling code inserted for allocation tracking; ii) **fast-call-profiling** represents the execution with all the profiling code but we did not turn on any method call tracking, i.e., no method call goes through the slow path (as described in Section 4.5.2); iii) **real-profiling** represents real benchmark execution, with all the profiling code; iv) **slow-call-profiling** represents the worst-case possible execution, with all profiling code, forcing all method calls to be tracked, i.e., all method calls go through the slow path (as described in Section 4.5.2)

We found the results in Figure 6.48 very interesting as different applications exercise the profiling code in different ways, resulting in different overheads for the same profiling code across different benchmarks. For example, for benchmarks such as `fop`, allocation profiling (the

Workload	LOC	AS	MC	PAS	PMC	Conflicts	NG2C	OLD size
Cassandra-WI	195 101	3 609	20 885	84	408	2	22	12 MB
Cassandra-RW	195 101	3 609	20 885	109	480	2	22	12 MB
Cassandra-RI	195 101	3 609	20 885	107	529	2	22	12 MB
Lucene	89 453	1 874	8 618	26	117	0	8	4 MB
GraphChi-CC	18 537	2 823	12 602	65	56	3	9	16 MB
GraphChi-PR	18 537	2 823	12 602	59	52	3	9	16 MB

Table 6.10: ROLP Profiling Summary

first bar from the left) leads to around 3% overhead while method call profiling leads to almost 10% overhead (difference between the first and second bars from the left). Other benchmarks reveal very different behavior, e.g., the `sunflow` benchmark, with high overhead for allocation profiling and almost zero overhead for method call profiling. It is also interesting to note that the **real-profiling** overhead is very close to the **fast-call-profiling** meaning that very few method calls were profiled in order to solve allocation context conflicts.

The left side of Table 6.9 presents the number of profiled method calls (**PMC**), the number of profiled allocation sites (**PAS**), and the number of conflicts found while executing each benchmark. From these results, we confirm that conflicts are not frequent. On the right side of Table 6.9, we present simulation results on what would be the expected throughput overhead for having 20% of all method calls being tracked (*P* from Section 4.5.5 is 20%) and how long this process would take in the worst-case scenario (we estimate this by taking the average time between two GC cycles). From these results, it is possible to observe that: i) conflict resolution overhead is never above 2% of additional throughput overhead, and ii) conflict resolution can take up to 520 seconds but for most benchmarks it could not take more than 2 minutes. It is still possible to reduce the duration by increasing *P* to higher percentages.

6.5.3 Large-Scale Application Profiling

This section summarizes the amount of profiling used when evaluating ROLP with the large-scale workloads, and also compares it to the amount of human-made code modifications necessary for NG2C. Table 6.10 presents a number of metrics for each workload: **LOC**, lines of code of the platform; **AS**/**MC**, number of allocation sites / method calls considered for profiling by the ROLP package filter; **PAS**/**PMC**, number of profiled allocation sites / method calls (i.e., allocation sites / method calls where profiling code was actually inserted); **Conflicts**, number of allocation context conflicts; **NG2C**, number of code locations that were changed to evaluate NG2C; **OLD size**, approximate memory overhead of the Object Life Time Distribution table (see Figure 4.10);

As described in Section 5.5.3, when running large-scale workloads (i.e., Cassandra, Lucene, and GraphChi), ROLP was launched with a filter to specify which packages should be profiled. The selected packages for profiling are the following:

- `org.apache.cassandra.{db,utils.memory}` for Cassandra;
- `org.apache.lucene.store` for Lucene;
- `edu.cmu.grapchi.{datablocks,engine}` for GraphChi.

These specific packages were selected because they are the ones that deal with most data in each platform. It is relevant to note that selecting these packages to consider for profiling is a much simpler task compared to understanding the average life time of objects allocated through particular allocation sites (which is required for NG2C).

From Table 6.10, three important points must be retained. Fist, looking at PAS and PMC, the number of profiled allocation sites and method calls is small (when compared with other values in the table, such as LOC, for example). This demonstrates that the profiling effort is greatly reduced by only profiling hot code locations, and by using optimizations such as avoiding inlined methods calls. Second, looking at OLD size, the memory overhead introduced to support profiling information does not exceed 16 MB, a reasonable memory overhead considering the performance advantages that can be achieved by leveraging the information in it. Finally, the number of allocation context conflicts does not exceed 3, showing that, despite using a weak hash construction (based on addition and subtraction of hashes), it is possible to achieve a low number of conflicts.

It is worthy to note that all the code changes done on the applications, which are needed to use NG2C, require either human knowledge (i.e., the programmer), or the use of a profiler (either offline, POLM2, or online, ROLP). When using ROLP, such changes are done automatically, i.e., the code is profiled and changes are done with no human intervention. ROLP additionally profiles other code locations (which are not used for NG2C), leading to additional improvements.

6.5.4 Pause Time Percentiles and Distribution

Figures 6.49 to 6.54 present the results for application pauses across all workloads: for NG2C with ROLP (ROLP), NG2C with POLM2 (POLM2), NG2C (with code annotations, NG2C), and G1. Pauses are presented in milliseconds and are organized by percentiles.

Compared to G1, ROLP significantly improves application pauses for all percentiles across all workloads. Regarding NG2C and POLM2, ROLP approaches the numbers achieved by

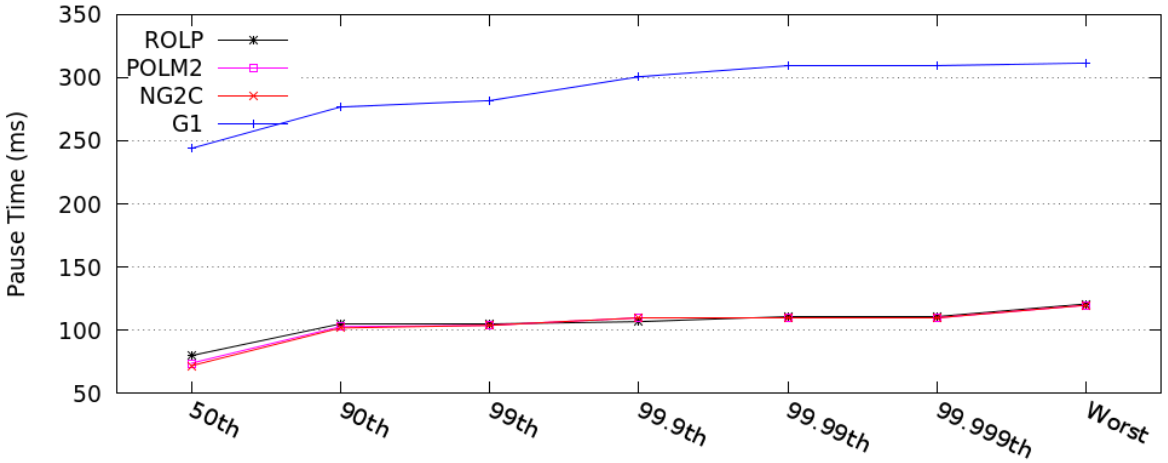

Figure 6.49: Pause Time Percentiles (ms) for Cassandra WI Workload

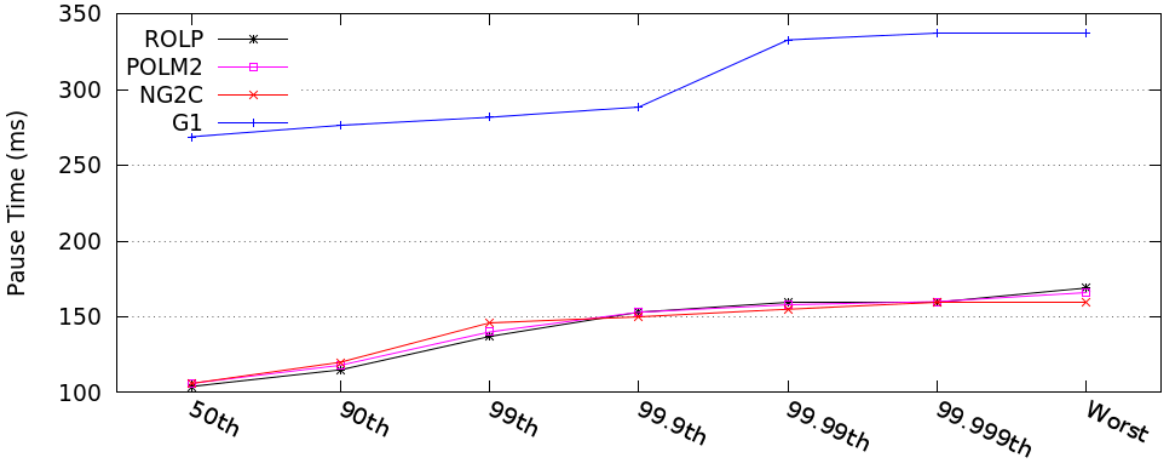

Figure 6.50: Pause Time Percentiles (ms) for Cassandra WR Workload

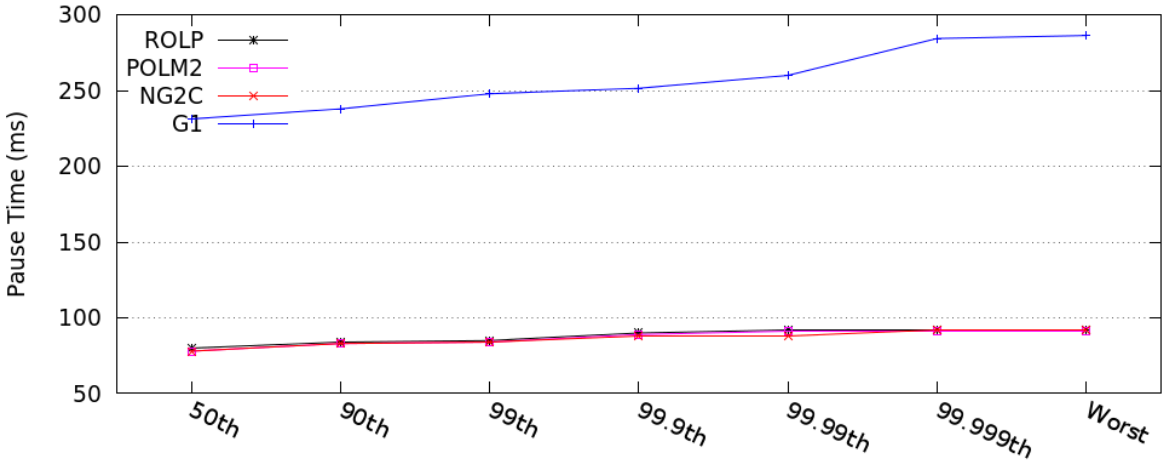

Figure 6.51: Pause Time Percentiles (ms) for Cassandra RI Workload

these two approaches for all workloads. From these results, the main conclusion to take is that ROLP can significantly reduce long tail latencies when compared to G1, the most advanced GC

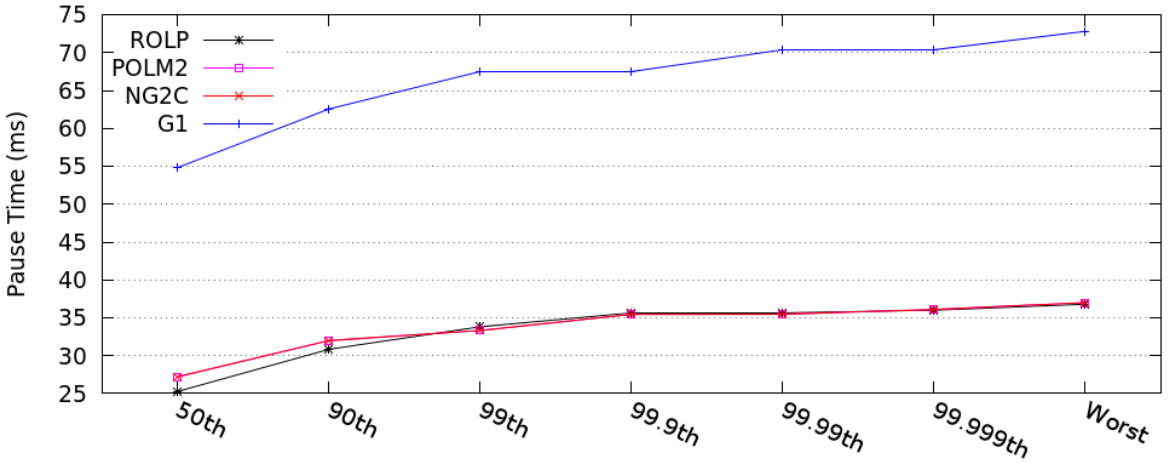

Figure 6.52: Pause Time Percentiles (ms) for Lucene Workload

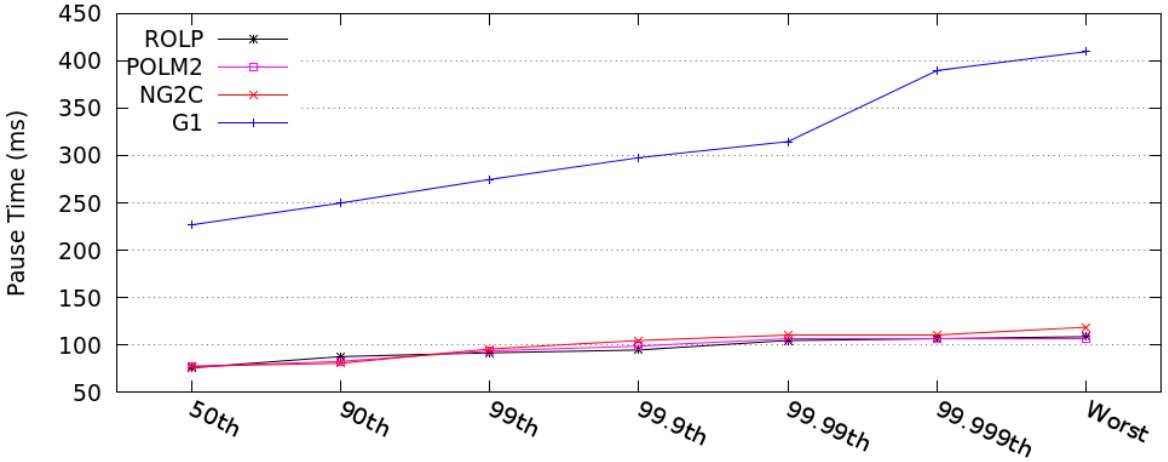

Figure 6.53: Pause Time Percentiles (ms) for GraphChi CC Workload

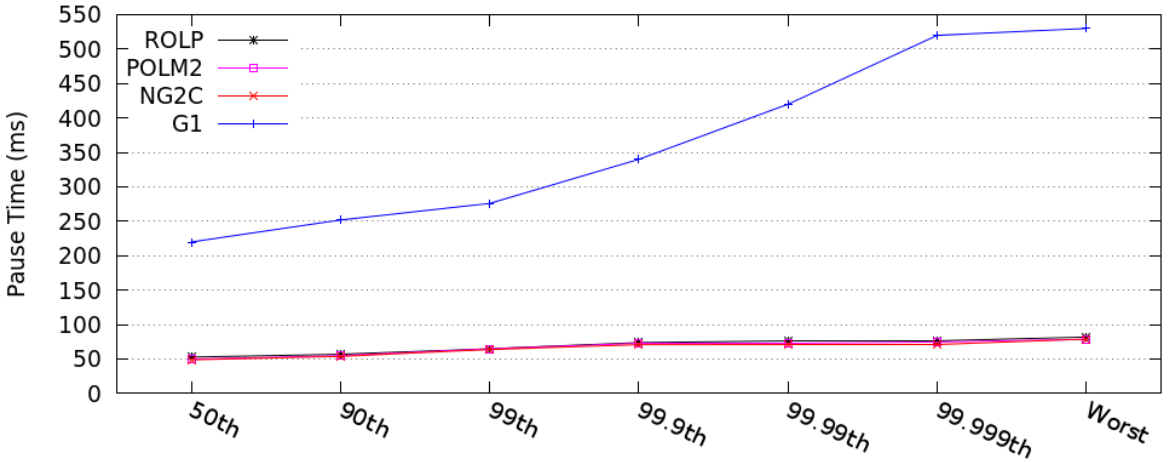

Figure 6.54: Pause Time Percentiles (ms) for GraphChi PR Workload

implementation in OpenJDK HotSpot; in addition, it can also keep up with NG2C which requires programming effort and knowledge, and also with POLM2 which requires offline profiling of the

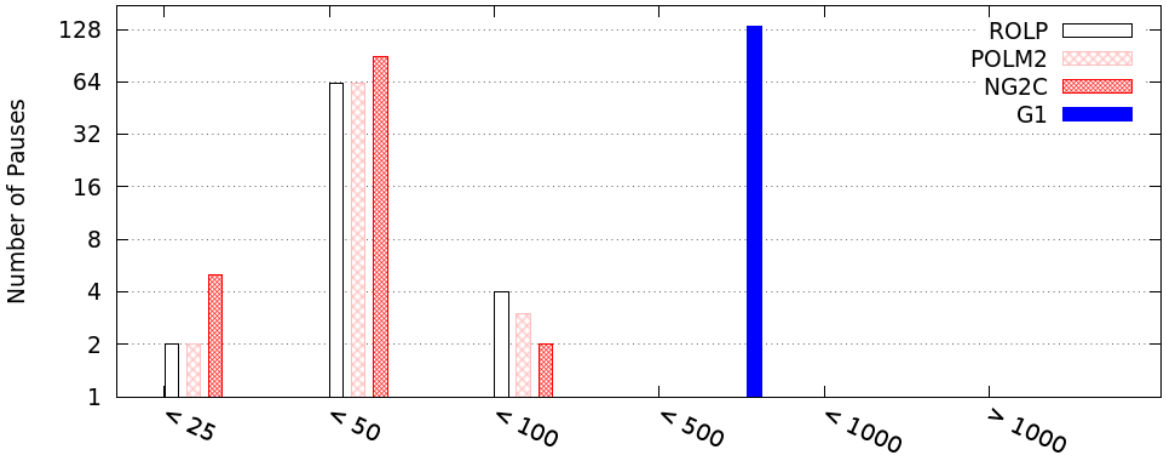

Figure 6.55: Application Pauses Per Duration Interval (ms) for Cassandra WI Workload

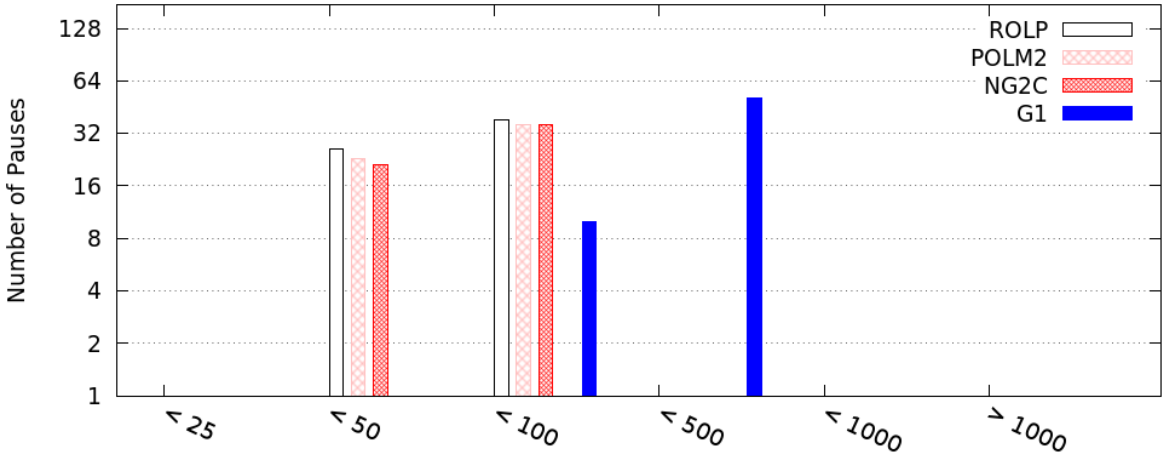

Figure 6.56: Application Pauses Per Duration Interval (ms) for Cassandra RW Workload

application.

Figures 6.55 to 6.60 present the number of application pauses that occur in each pause time interval. Pauses with shorter durations appear in intervals to the left while longer pauses appear in intervals to the right. In other words, the less pauses to the right, the better.

ROLP presents significant improvements regarding G1, i.e., it results in less application pauses in longer intervals, across all workloads. When comparing ROLP with NG2C and POLM2, all three solutions present very similar pause time distribution.

In sum, ROLP allows NG2C to reduce application pauses by automatically pretenuring objects from allocation contexts that tend to allocate objects with longer life times. When compared to G1, ROLP can greatly reduce application pauses and object copying within the heap. Once again, we can say that when compared to NG2C and POLM2, ROLP presents equivalent performance without requiring programmer effort and knowledge, or offline profiling, and which also adapts to dynamic workloads.

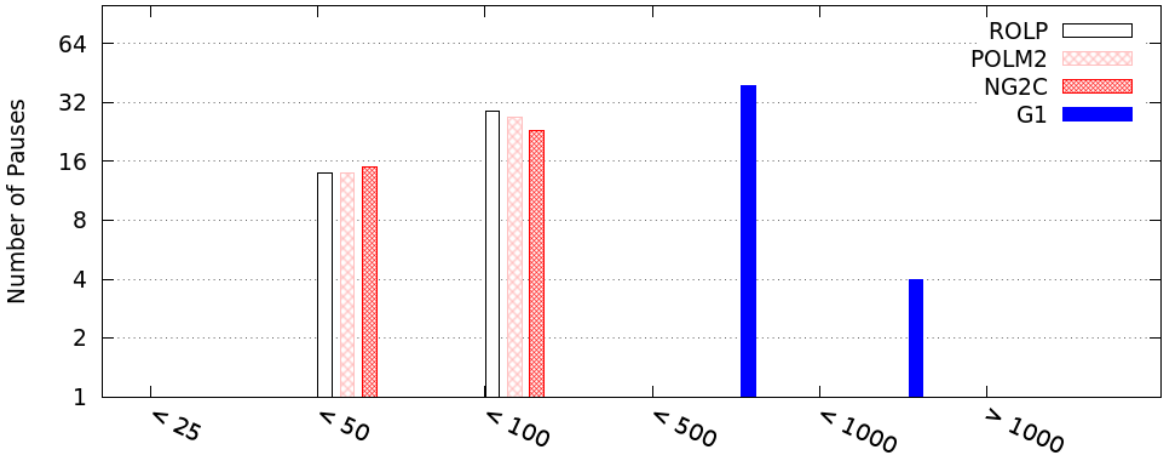

Figure 6.57: Application Pauses Per Duration Interval (ms) for Cassandra RI Workload

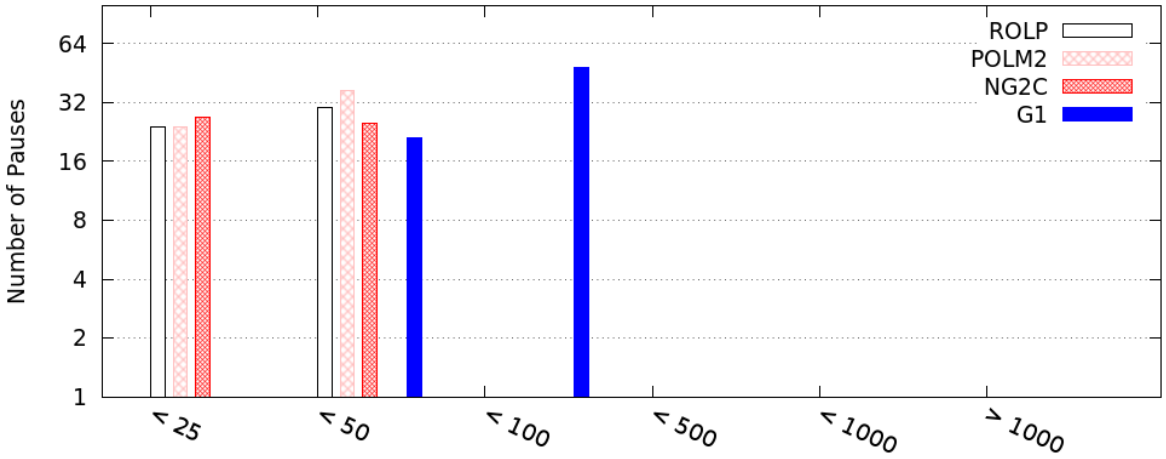

Figure 6.58: Application Pauses Per Duration Interval (ms) for Lucene Workload

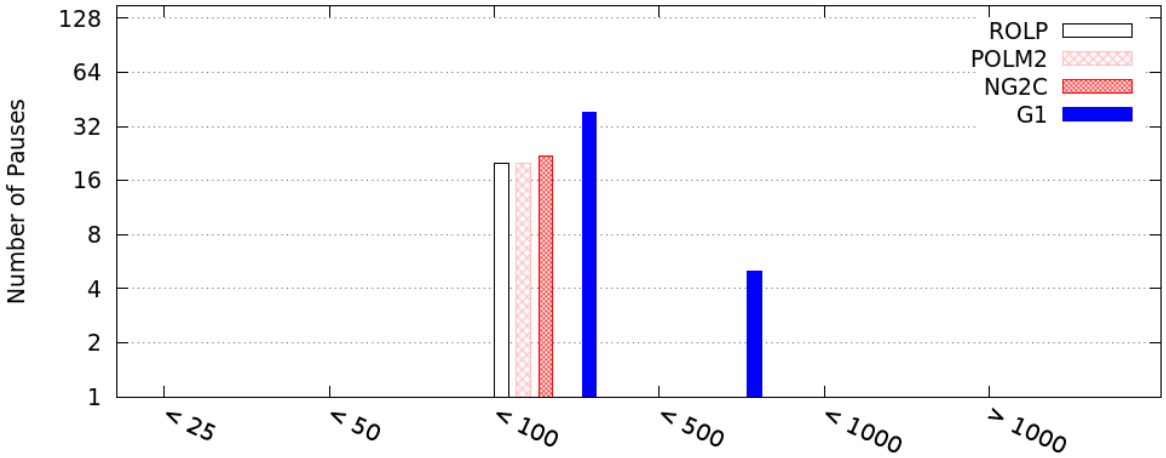

Figure 6.59: Application Pauses Per Duration Interval (ms) for GraphChi CC Workload

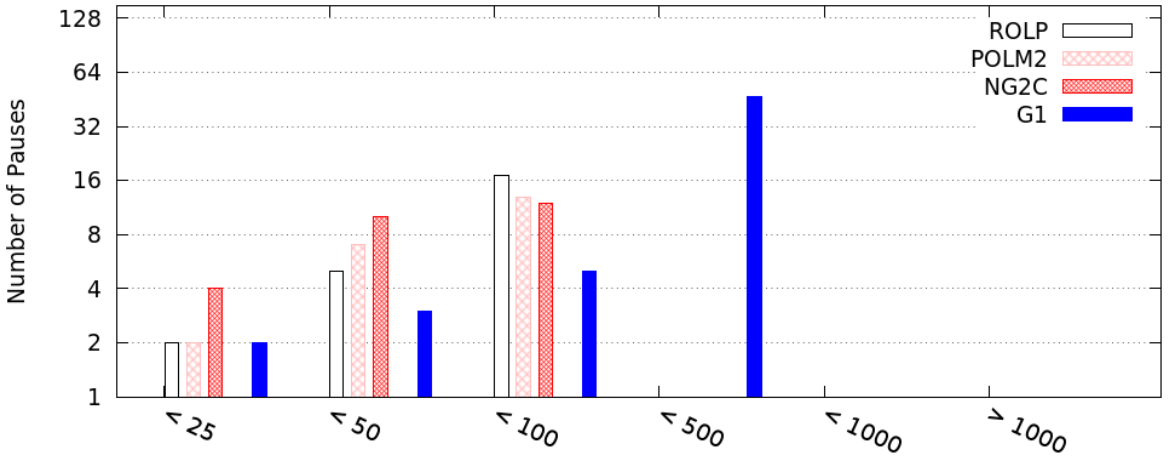

Figure 6.60: Application Pauses Per Duration Interval (ms) for GraphChi PR Workload

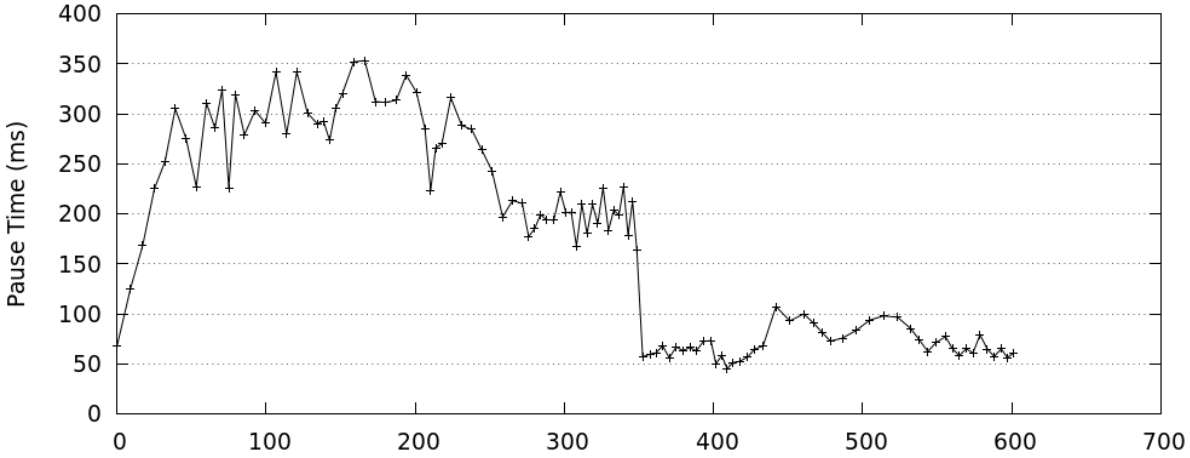

Figure 6.61: Cassandra WI Warmup Pause Time (ms)

6.5.5 Warmup Pause Times, Throughput and Memory Usage

This section shows results on application warmup pause times, throughput, and max memory usage. Note that application warmup happens when the workload changes and ROLP is still detecting (i.e., learning) the life time of objects. Clearly, such time interval should be the minimum possible. Thus, the goal of this section is to: i) see how the learning curve of ROLP affects pause times during warmup and how long does it take; i) show that ROLP does not inflict a significant throughput overhead due to its profiling code; and iii) show that ROLP does not negatively impact the max memory usage.

Figure 6.61 shows the Cassandra WI warmup pause times for the first 600 seconds of the workload. Pause times during the warmup phase can be divided into three parts. The first part spans from the beginning to around 250 seconds. During this part of the execution, no information is given to NG2C since ROLP is still gathering information regarding objects'

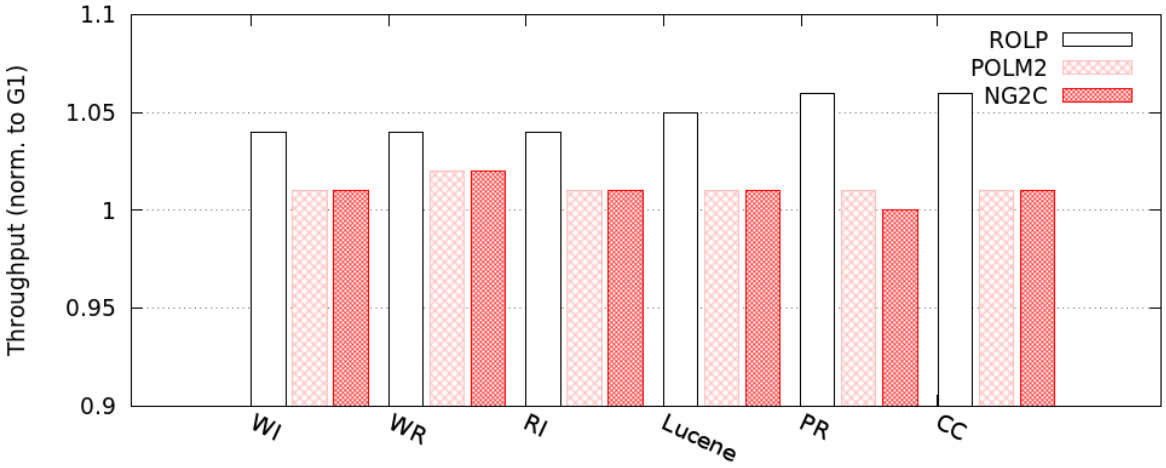

Figure 6.62: Average Throughput normalized to G1

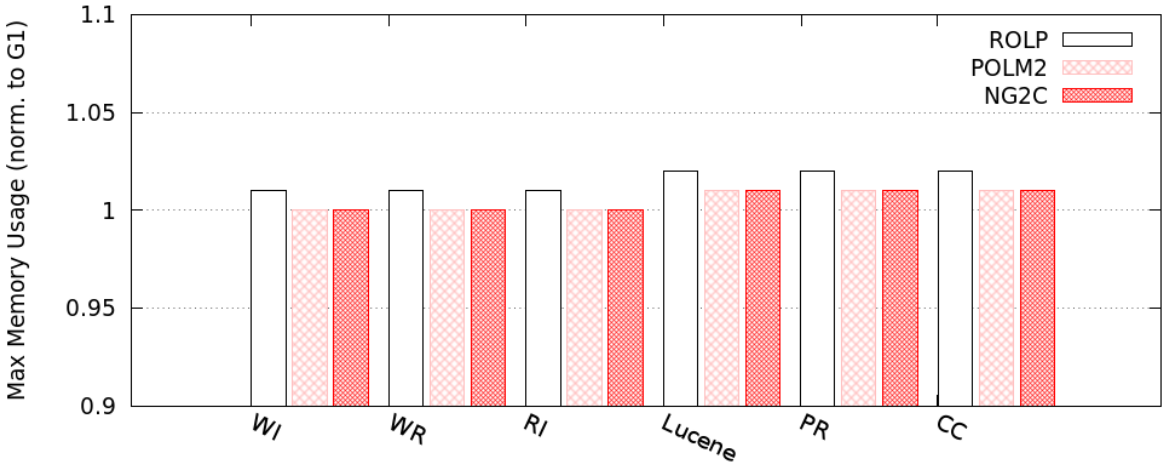

Figure 6.63: Max Memory Usage normalized to G1

life times. Around second 250 (until second 350), ROLP already performed some life time estimations, and NG2C starts pretenuring application objects resulting in reduced pause times. Finally, the third part of the warmup starts around the second 350 when NG2C receives more information profiling information. At this point, ROLP also decides to turn off survivor tracking (see Section 5.5.4) to further reduce pause times. In short, ROLP takes about 350 seconds to stabilize the profiling information in Cassandra. In a real production environment, in which such workloads can run for days, 350 seconds represents a very small time to stabilize the system given its performance benefits. It is important to note that this number of seconds depends on the frequency of GCs. The more GCs the application trigger, the faster ROLP will provide information to NG2C (as explained in Section 4.5.3).

With regards to throughput (see Figure 6.62) and max memory (see Figure 6.63), ROLP presents a negligible throughput decrease, less than 5% (on average) for most workloads, compared to G1. Only for GraphChi workloads, ROLP presents an average throughput overhead

of 6% for both PR and CC). We consider this a negligible throughput overhead considering the great reduction in application long tail latencies. Memory usage also shows a negligible overhead of ROLP compared to both G1, NG2C, and POLM2.

6.6 Vertical Scaling Evaluation

This final evaluation section presents results for our dynamic vertical scalability solution described in Section 4.6. 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 6.6.2); ii) show that this reduction in the JVM memory footprint does not impose a significant performance overhead for applications (see Section 6.6.3); iii) show that the memory overhead (for holding large GC data structures) associated to having a very large `MaxMemory` limit is negligible (see Section 6.6.4); iv) estimate how much cloud users and providers can save by allowing JVM applications to scale memory vertically (see Section 6.6.5).

In each experiment, we show results for both our implementations (G1/NG2C and PS) and their respective original implementations. For simplicity our plots are labeled and presented 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).

6.6.1 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.

In order to produce workloads to exercise our solution, we take advantage of the widely used and studied DaCapo 9.12 benchmark suite [13]. 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 6.11 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:

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

Table 6.11: DaCapo Benchmarks

`CurrentMaxMemory` (i.e., memory that the application can use), `MaxOverCommittedMemory` (i.e., maximum amount of committed memory that is not being used by the application that does not trigger heap resizing), and `MinTimeBetweenGCs` (i.e., minimum time between GCs and heap resizing operations).

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 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. 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 6.6.4, all experiments are configured with `CurrentMaxMemory` equal to `MaxMemory`.

The `MaxOverCommittedMem` is set to either half or quarter of the heap size. We found that these values provide a good memory scalability while imposing a negligible throughput overhead (this trade-off is further discussed in Section 6.6.3). `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 depend on the billing period of the cloud provider (hourly, daily, etc).

6.6.2 Dynamic Memory Scalability

This section presents results on how much can an application memory footprint be reduced by employing the heap sizing strategy proposed in Section 4.6.2. This footprint change is presented from two perspectives: i) the container memory usage (see Figure 6.64), and ii) the

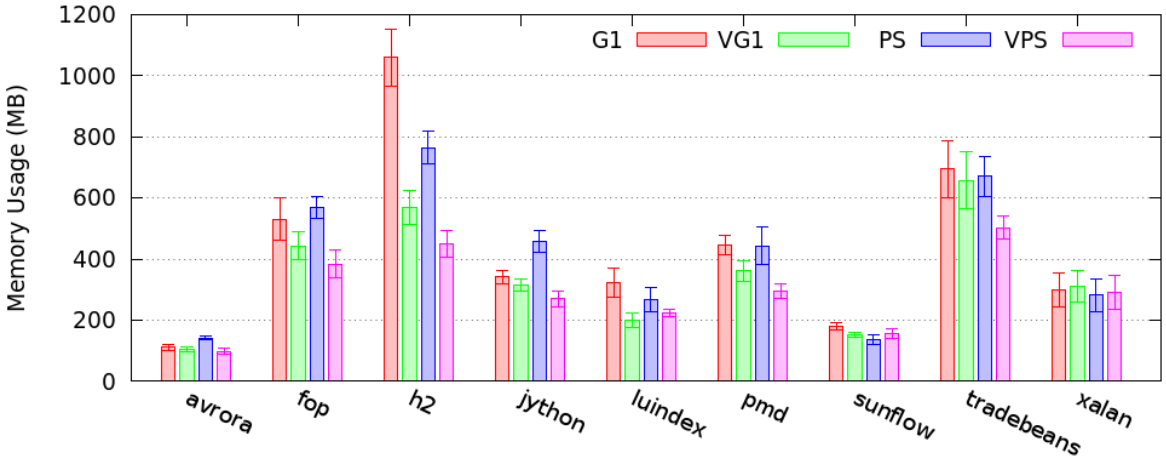

Figure 6.64: Container Memory Usage (MB)

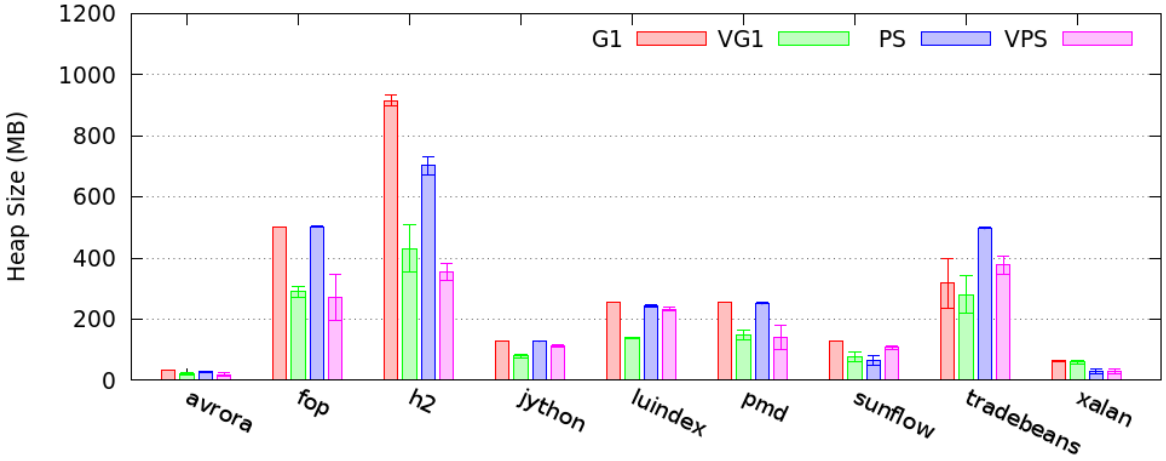

Figure 6.65: JVM Heap Size (MB)

JVM heap size or committed memory (see Figure 6.65). Both figures present the average and standard deviation 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 aggressive mutators, i.e., application that allocate memory more quickly. This leads the collector to pre-allocate more memory in the hope that the mutator will use it in the future. Taking the `h2` benchmark as 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 6.65). Both plots are highly correlated as the JVM heap

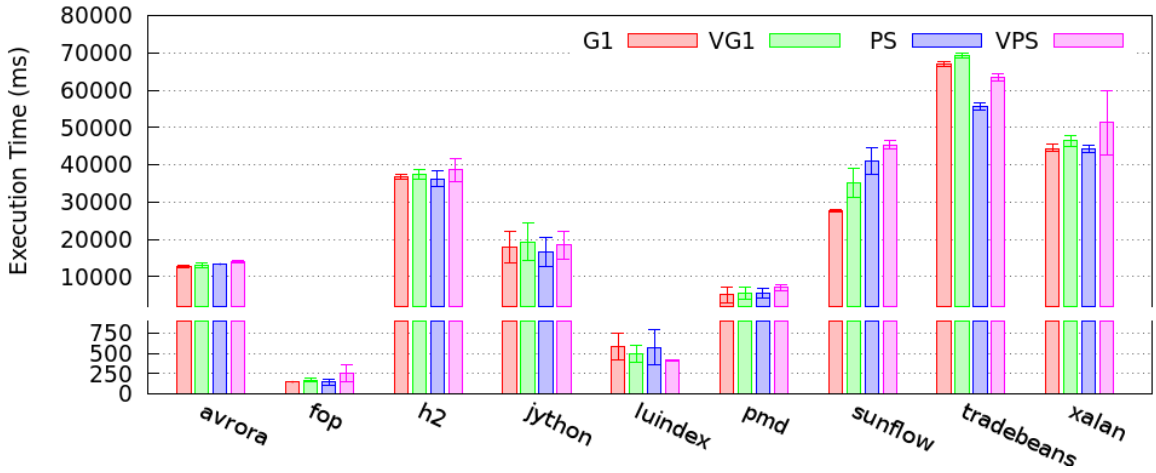

Figure 6.66: Execution Time (ms)

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.

6.6.3 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 resizing approach is enforced.

Figure 6.66 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 6.11), in particular the value of the parameter `MaxOverCommittedMemory`. Figures 6.67 and 6.68 present VG1 throughput versus memory trade-off 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 reduces.

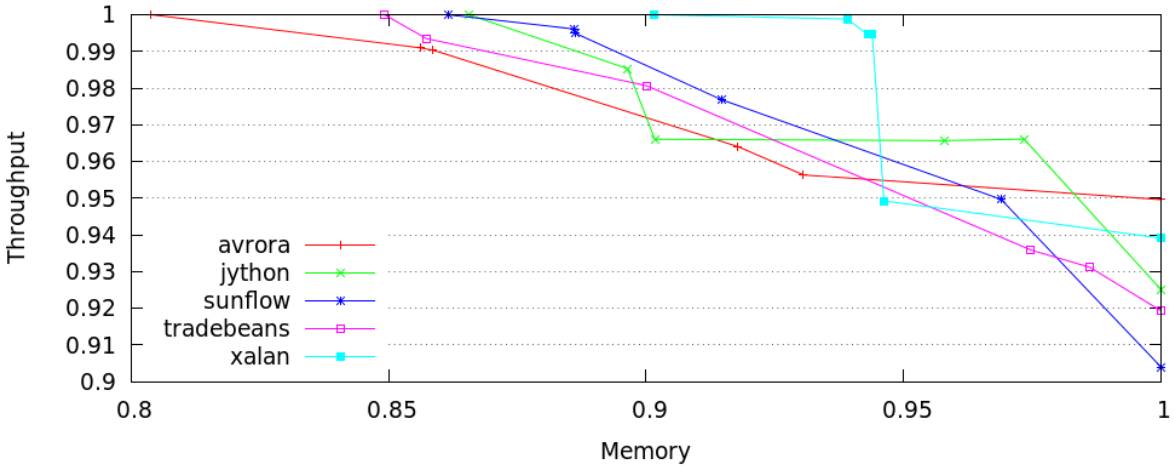

Figure 6.67: Throughput vs Memory Trade-off (a)

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 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 conclude that we can reduce the average memory utilization by 30 % at a 2 % throughput overhead.

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 most benchmarks, with less than 10 % of throughput overhead, it is possible to reduce the memory footprint by up to least 30 %. In sum, different applications might have different throughput/memory trade-offs and therefore, one needs to identify, for each application, the best trade-off, i.e., how much throughput to sacrifice for which amount of reduced memory footprint.

6.6.4 Internal Data Structures Overhead

As discussed in Section 4.6, 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.

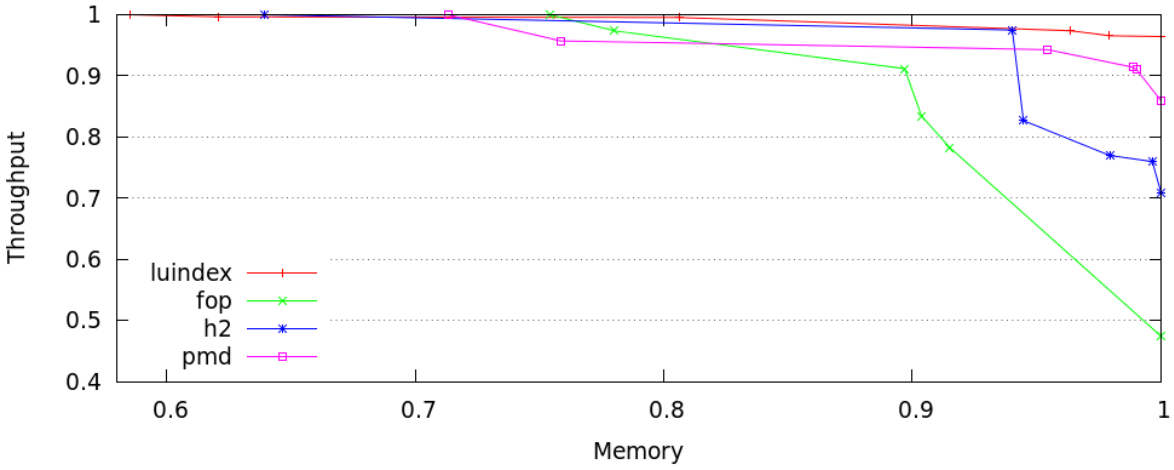

Figure 6.68: Throughput vs Memory Trade-off (b)

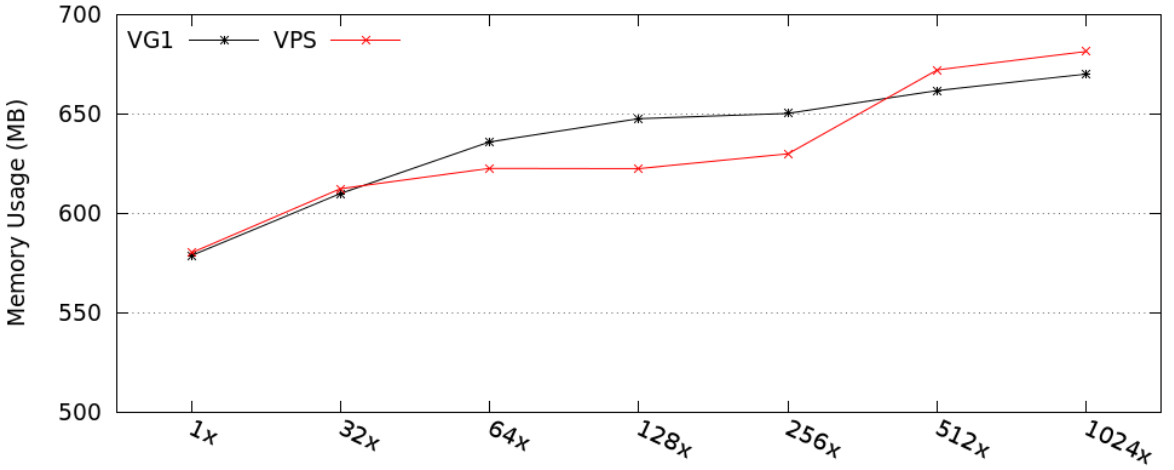

Figure 6.69: `h2` Container Used Memory (MB) for Different Max Heap Limits

Figure 6.69 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 1024 MB (this was also the value used in previous experiments for this benchmark). This experiment exercises values of `MaxMemory` starting at 1x the `CurrentMaxMemory` (1 GB) 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 a significant increase memory footprint. In `h2`, setting a heap max size 32x larger compared to the smallest memory footprint with highest throughout only adds 31.3 MB to the container. In other words, increasing the `MaxMemory` by 32 GB results in 31.3 MB of increased container memory usage. We do not show the results for this experiment with other benchmarks because the results are exactly the same. Besides, 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 exactly the same. In conclu-

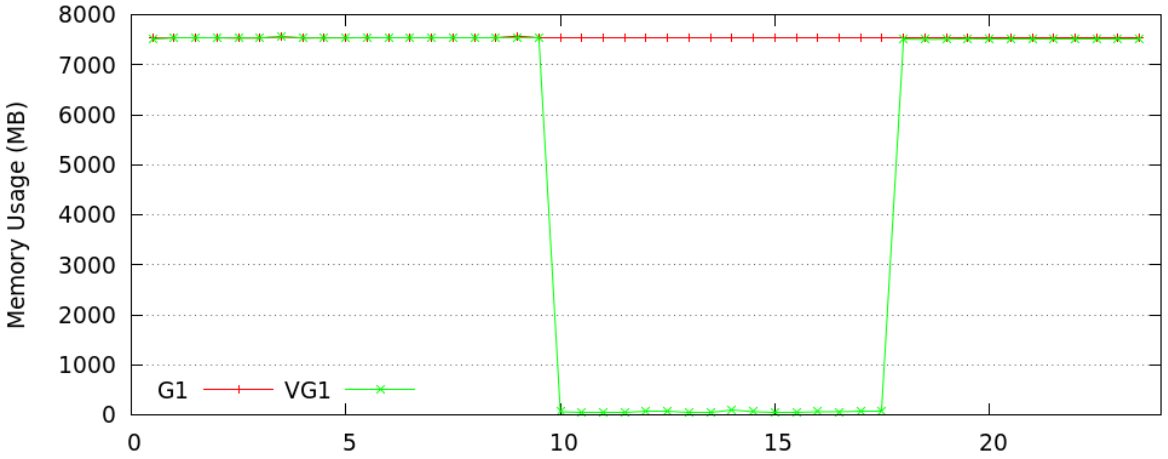

Figure 6.70: Tomcat Memory Usage (MB) during 24 hours

Table 6.12: 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%

sion, the small amount of footprint lost by setting a very conservative `MaxMemory` is far inferior to the amount of memory recovered by periodic heap sizing checks.

6.6.5 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 web-server 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 6.70 presents a plot showing a Tomcat web-server 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 6.70) 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 6.12, 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 6.12, 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), and since memory is the limiting factor for oversubscribing [73], it is possible to collocate instances and reduce up to 64x the amount of hardware used to run the same instances.

6.7 Summary

In this evaluation chapter, we have presented an extensive set of performance experiments to validate that contributions described in this work. In particular, we evaluated the performance of ALMA JVM live migration algorithm and, as results show, it outperforms current JVM live migration alternatives. This mostly comes from the fact that ALMA only migrates the JVM process (and ignores all the other environment state such as unreachable application data, OS kernel and other processes) and also from the fact that ALMA provides an efficient technique to determine which parts of memory to collect before performing the snapshot to migrate.

After evaluating ALMA, we moved into evaluating our garbage collection improvements, NG2C, POLM2, and ROLP. NG2C greatly reduces application pause times compared to current collectors available in OpenJDK. However, it needs application code changes that are not trivial to achieve. POLM2 solves this problem by relying on a profiling phase used to learn how the application allocates objects. The knowledge acquired during the profiling phase can, as seen through results, replace application programmer knowledge. However, POLM2 is not resilient to workload changes as it profiles the application towards a single workload. To solve this problem, we then propose ROLP which runs inside the JVM and automatically produces profiling information that replaces programmer knowledge and is resilient to workload shifts.

With regards to application pause times, all three solutions can significantly reduce application pause times compared to other collectors already available in OpenJDK. As for throughput, both NG2C and POLM2 produce no throughput overhead while ROLP leads to a very

small throughput overhead (due to the profiling code inserted during JIT compilation). Finally, with regards to user effort, ROLP is the solution with less user effort, followed by POLM2, and NG2C. Also note that the less user effort required also means less control and therefore, some advanced programmers, might prefer having more control over how pretenuring is being used.

It is important not to confuse the goal of NG2C, POLM2, and ROLP, with the goals of concurrent compaction collectors (such as C4). For the later ones, their goal is to achieve ultra-low pause times (less than 10 ms) at the cost of a moderate to high throughput overhead. Our approach however provides a different trade-off, where we reduce significantly application pause times with zero (for NG2C and POLM2) or very reduced (for ROLP) application throughput overhead.

The evaluation chapter closes with an evaluation of our dynamic vertical scalability solution which allows JVM applications to freely scale memory. This is specially important in a cloud setting (as previously described in Section 3.3). From our results, we conclude that it is possible to significantly reduce the cost of cloud hosting by dynamically scaling the amount of memory that is being used by the JVM.

Chapter 7

Conclusions and Future Work

Having presented an extensive set of performance experiments that confirm the performance improvements of the proposed algorithms, we now dedicate this final chapter summarize and conclude this work, followed by future work ideas.

7.1 Conclusions

This document presented novaVM, an enhanced Java Virtual Machine for Big Data applications. The main motivation for this work comes from the current need for Big Data applications in many areas ranging from scientific experiments to online credit card transaction validation. In addition, Big Data applications are being increasingly ran on top of runtime systems (such as OpenJDK HotSpot) which provide easy programming abstractions, portability, automatic memory management, among others. Therefore, we identify and propose solutions for some of the problems Big Data applications encounter when running on top of runtime systems, in particular, the Java Virtual Machine.

Throughout this work we addressed a number of problems (presented in Chapter 1), analyzed current solutions for each of the proposed problems and identified opportunities for improvements regarding current state of the art (in Chapter 3). Then, we presented the design and implementation of novaVM (Chapters 4 and 5), and finally evaluated its performance and compared it with previous solutions (in Chapter 6).

In sum, we presented ALMA (see Section 4.2), a GC-aware live Checkpoit/Restore tool for migrating JVMs. This solution solves Problem 1 (the need to quickly recover from failed nodes or to spawn more nodes to accommodate new workload demands). By taking advantage of internal GC information, ALMA is able to significantly reduce the amount of data to migrate, thus improving the performance of live migration. Compared to current approaches, ALMA is

able to reduce the application downtime, total migration time, and network bandwidth required for migration.

The long tail latency problem (Problem 2) is solved by the combination of three proposed algorithms, a new GC algorithm which allows objects to be allocated in multiple allocation spaces (generations), combined with a profiler that maps application allocations into allocation spaces. By combining NG2C (Section 4.3) with one of the proposed profilers, POLM2 (Section 4.4) or ROLP (Section 4.5), the resulting system is able to reduce application pause times by grouping objects with similar life times close to each other. By doing so, fragmentation is dramatically reduced, resulting in shorter pause times. When compared to current OpenJDK GC algorithms, our proposed solution is able to significantly reduce application pause times with negligible throughput and memory footprint impact.

Finally, we proposed a Dynamic Vertical Scalability (Section 4.6) algorithm to allow the memory of the JVM to scale vertically and therefore, to improve the way the JVM fits in the virtualization stack (Problem 3). This is of crucial importance for cloud scenarios where reducing the amount of unused memory in the JVM is beneficial for both cloud providers and customers. In addition, we were able to show that, with real-world workloads, our solution is able to substantially reduce the cost of cloud hosting.

To conclude, we developed a set of solutions for current real-world problems. Throughout this process we published a number of research papers in international conferences and journals, and also contributed with software (in the form of patches) derived from the implementation of this work to very relevant opensource projects such as CRIU and OpenJDK HotSpot JVM. In addition, we also leave several future work ideas, which are presented next.

7.2 Future Work

Through recent years, we have seen an increase in the number of languages that run on top of runtime systems. JavaScript, Java, C#, Scala, Python, and Go are just a few examples of widely adopted managed languages that run on top of such runtimes. The extensive use of managed languages reveals that application developers want to take advantage of all the benefits of using a runtime system, and also shows that current runtimes' design is mature, providing competitive performance when compared to traditional languages such as C and C++. Therefore, we foresee that runtime system utilization will continue to grow in the future, suggesting the need for more research in this area (such as the one presented in this work).

In previous chapters we presented a set of new algorithms that improve runtime systems used to run Big Data applications. To conclude our research contribution, we now elaborate on

how this work could be expanded/used together with new research community developments and also other possible research directions on how to improve the performanfe of Big Data applications running on runtime environments.

7.2.1 Latency vs Throughput vs Footprint

Many research works, this one included, have shown that memory management algorithms, GC algorithms in particular, can be tuned in order to prioritize specific performance metrics such as: latency, throughput, and memory utilization (footprint). In addition, with the recent development of new GC algorithms providing ultra-low latency (C4[113], Shanendoah[1], and ZGC[2]), the memory management landscape becomes more complex. In particular, selecting the correct GC for a specific application is not trivial since each GC provides different trade-offs.

In sum, there is currently a need to study how current GCs (including NG2C) perform with regards to these metrics to better understand their trade-offs. An example of such trade-off is the need of higher footprint and lower throughput in order to guarantee lower pause time (this can be observed in C4, Shanendoah, and ZGC).

Findings on how current GCs behave, for example, with smaller footprints, could impact both ALMA and our Dynamic Vertical Scalability algorithms, which try to reduce the amount of memory used before a JVM live migration or during periods of less application activity.

7.2.2 Ultra-Low Pause Time GCs

Ultra-low pause time GC algorithms (such as C4, Shanendoah, and ZGC) are becoming more and more common as many applications require low pause times. This leads to several interesting questions.

First, low pause times are achieved at the cost of barriers that stop the application if some GC work is concurrently executing over the object that the application needs to access. The throughput cost of these barriers is high, at least 15% to 30% for different types of applications. One important research challenge is to identify opportunities for disabling such barriers. Currently, these barriers are always turned on, meaning that an application will suffer throughput degradation even if the collector is not working. This is clearly sub-optimal and being able to disable such barriers when not needed would lead to great improvements in current collectors' performance.

Second, even if compaction (i.e., object copying) is done concurrently, it still produces a

[1] https://wiki.openjdk.java.net/display/shenandoah/Main
[2] https://wiki.openjdk.java.net/display/zgc/Main

negative impact on the application because it might have to wait for some object to be copied or memory bandwidth might become the bottle-neck of the application performance. To avoid having to compact objects, these collectors should also use profilers such as ROLP to allocate objects with similar life times close to each other. Hence, it would be interesting to port ROLP or POLM2 into a concurrent compaction collector.

Third, other ultra-low pause time GCs have taken a different path and opted for not supporting object compaction. This simplifies the implementation and less barriers are necessary. However, it leads to higher memory fragmentation, as objects cannot be moved. It is interesting to note that programs written in languages such as C and C++ also suffer from the same problem. Nevertheless, the Go[3] runtime is a well-known example of this approach. One question that remains to be answered is how significant is fragmentation for the performance of Big Data applications.

7.2.3 Just-In-Time Compilation

Just-In-Time compilers are also major players in the performance of applications running on runtime systems. Very recently, a new JIT compiler, Graal[4], has been released with very interesting properties, namely a very strong escape analysis that can be used to improve GC algorithms. Another great advantage of this new JIT compiler is that it is purely written in Java, allowing quick and easy development of experimental ideas.

In sum, this new compiler can also be a baseline for future improvements on ROLP. For example, to try new ways of profiling or sample application allocation in order to mitigate the application throughput overhead.

7.2.4 Object Graph Tracing for Large Heaps

Tracing is a fundamental step recent garbage collectors as it is required to identify live objects. From time to time, the collector will trace the whole application object graph even if this object graph remains unchanged since the last tracing iteration. For example, if an application is mostly writing in some memory regions, other memory regions will remain unchanged and therefore, local object liveliness will remain the same. If a tracing cycle is triggered, the collector will trace through unchanged memory regions eventhough the graph is unchanged and therefore, live objects are exactly the same.

Garbage First (the current HotSpot default collector) provides a solution to this problem

[3] https://golang.org
[4] https://www.graalvm.org/

(useless tracing) by keeping remembered sets between each memory regions. This allows one to perform mixed collections, ones that collect memory regions from multiple spaces without requiring a full tracing cycle. However, this method is costly as it introduces a lot of remembered set updates (which impact the throughput).

A possible research hypothesis is to use the underlying host OS to identify unmodified memory pages that could be skipped during a full tracing cycle. This technique is similar to ALMA's technique to perform incremental snapshots by using the Linux kernel dirty-bit. This technique requires less write barriers to update remembered sets and therefore might be an interesting future research goal.

7.2.5 Accelerated JVM

Nowadays, the JVM runs in an isolated environment, with very limited access to information regarding the external environment, such as the capabilities of the hosting environment. Due to the number of platforms that are currently running on top of JVMs, it would be interesting to analyze and propose ideas for reaching devices other than CPU and RAM such as accelerators (FPGAs, GPUs, etc).

Similarly to what TensorFlow provides in a Python environment, the JVM could also provide such support for accelerated computations for all the languages that it already runs (Java, Scala, Groovy, etc). This would allow many applications to be accelerated (i.e., to run in accelerators) with little to no changes and with considerable advantages in the performance.

Allowing the JVM to harness different execution models and architectures raises multiple interesting questions such as how to efficiently generate code for such devices or how to manage memory across multiple devices.

Bibliography

[1] R. Akerkar. *Big data computing*. CRC Press, 2013.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[3] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, PP(99): 1–1, 2017. ISSN 1939-1374. doi: 10.1109/TSC.2017.2711009.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, et al. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

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

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

[7] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21 (4):280–294, Apr. 1978. ISSN 0001-0782. doi: 10.1145/359460.359470. URL `http://doi.acm.org/10.1145/359460.359470`.

[8] G. Banga, P. Druschel, and J. C. Mogul. 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, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL `http://dl.acm.org/citation.cfm?id=296806.296810`.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth*

ACM Symposium on Operating Systems Principles, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945462. URL http://doi.acm.org/10.1145/945445.945462.

[10] P. B. Bishop. Computer systems with a very large address space and garbage collection. Technical report, DTIC Document, 1977.

[11] S. Blackburn, P. Cheng, and K. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 137–146, May 2004. doi: 10.1109/ICSE.2004.1317436.

[12] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: 10.1145/949305.949336. URL http://doi.acm.org/10.1145/949305.949336.

[13] 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. 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*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[14] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[15] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1151–1162. IEEE, 2011.

[16] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[17] D. Box and T. Pattison. *Essential. Net: the common language runtime*. Addison-Wesley Longman Publishing Co., Inc., 2002.

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

[19] 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.

[20] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of java applications. *ACM Trans. Program. Lang. Syst.*, 28(5):908–941, Sept. 2006. ISSN 0164-0925. doi: 10.1145/1152649.1152652. URL http://doi.acm.org/10.1145/1152649.1152652.

[21] R. Bruno and P. Ferreira. Polm2: Automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 147–160, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4720-4. doi: 10.1145/3135974.3135986. URL http://doi.acm.org/10.1145/3135974.3135986.

[22] R. Bruno and P. Ferreira. A study on garbage collection algorithms for big data environments. *ACM Comput. Surv.*, 51(1):20:1–20:35, Jan. 2018. ISSN 0360-0300. doi: 10.1145/3156818. URL http://doi.acm.org/10.1145/3156818.

[23] R. Bruno, L. P. Oliveira, and P. Ferreira. Ng2c: Pretenuring garbage collection with dynamic generations for hotspot big data applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 2–13, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5044-0. doi: 10.1145/3092255.3092272. URL http://doi.acm.org/10.1145/3092255.3092272.

[24] R. Bruno, P. Ferreira, R. Synytsky, T. Fydorenchyk, J. Rao, H. Huang, and S. Wu. Dynamic vertical memory scalability for openjdk cloud applications. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2018, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5801-9. doi: 10.1145/3210563.3210567. URL https://doi.org/10.1145/3210563.3210567.

[25] R. Bryant, R. H. Katz, and E. D. Lazowska. Big-data computing: Creating revolutionary breakthroughs in commerce, science and society, 2008.

[26] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso. Dynamic management of virtual infrastructures. *Journal of Grid Computing*, 13(1):53–70, Mar 2015. ISSN 1572-9184. doi: 10.1007/s10723-014-9296-5. URL https://doi.org/10.1007/s10723-014-9296-5.

[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[28] K. Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[29] 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.

[30] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754181. URL http://doi.acm.org/10.1145/2754169.2754181.

[31] N. Cohen and E. Petrank. Data structure aware garbage collector. In *ACM SIGPLAN Notices*, volume 50, pages 28–40. ACM, 2015.

[32] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, Dec. 1960. ISSN 0001-0782. doi: 10.1145/367487.367501. URL http://doi.acm.org/10.1145/367487.367501.

[33] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. ISBN 1-58113-011-2. URL http://dl.acm.org/citation.cfm?id=266989.267068.

[34] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[36] 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.

[37] 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.

[38] D. Dice, M. Moir, and W. Scherer. Quickly reacquirable locks, Oct. 12 2010. URL `https://www.google.ch/patents/US7814488`. US Patent 7,814,488.

[39] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11): 966–975, 1978.

[40] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1):151–174, 2000. ISSN 0018-8670. doi: 10.1147/sj.391.0151.

[41] J. Dittrich and J.-A. Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[42] 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.

[43] 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. ISBN 1-58113-199-2. doi: 10.1145/349299.349336. URL `http://doi.acm.org/10.1145/349299.349336`.

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

[45] 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. doi: 10.1109/IC2E.2014.41.

[46] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[47] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction*, pages 82–93. Springer, 2000.

[48] L. George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[49] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[50] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *ACM SIGPLAN Notices*, volume 48, pages 229–240. ACM, 2013.

[51] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: A garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[52] 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.

[53] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[54] J. Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[55] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 325–340, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: 10.1109/CGO.2007.20. URL `http://dx.doi.org/10.1109/CGO.2007.20`.

[56] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment*, 4(11):1111–1122, 2011.

[57] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065028. URL http://doi.acm.org/10.1145/1065010.1065028.

[58] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard. Waste not, want not: Resource-based garbage collection in a shared environment. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 65–76, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993487. URL http://doi.acm.org/10.1145/1993478.1993487.

[59] 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.

[60] 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. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741950. URL http://doi.acm.org/10.1145/2741948.2741950.

[61] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *Memory Management*, pages 388–403. Springer, 1992.

[62] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.

[63] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[64] 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.

[65] R. Jones and A. C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 129–138. IEEE, 2005.

[66] R. Jones and C. Ryder. Garbage collection should be lifetime aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, 2006.

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

[68] R. E. Jones and C. Ryder. A study of java object demographics. In *Proceedings of the 7th international symposium on Memory management*, pages 121–130. ACM, 2008.

[69] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 152–162, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029892. URL `http://doi.acm.org/10.1145/1029873.1029892`.

[70] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 473–484, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451168. URL `http://doi.acm.org/10.1145/2451116.2451168`.

[71] 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. URL `https://www.usenix.org/conference/atc16/technical-sessions/presentation/kashyap`.

[72] 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.

[73] S. Kim, H. Kim, J. Lee, and J. Jeong. Group-based memory oversubscription for virtualized clouds. *J. Parallel Distrib. Comput.*, 74(4):2241–2256, Apr. 2014. ISSN 0743-7315. doi: 10.1016/j.jpdc.2014.01.001. URL `http://dx.doi.org/10.1016/j.jpdc.2014.01.001`.

[74] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[75] 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.

[76] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965. ISSN 0001-0782. doi: 10.1145/365628.365655. URL http://doi.acm.org/10.1145/365628.365655.

[77] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751.

[78] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[79] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[80] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[81] 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.

[82] 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.

[83] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.

[84] 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.

[85] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9 (12):936–947, Aug. 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994513. URL http://dx.doi.org/10.14778/2994509.2994513.

[86] P. Lu and K. Shen. 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, pages 3:1–3:15, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6. URL http://dl.acm.org/citation.cfm?id=1364385.1364388.

[87] C. Lynch. Big data: How do your data grow? *Nature*, 455(7209):28–29, 2008.

[88] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 457–471. ACM, 2016.

[89] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[90] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., Greenwich, CT, USA, 2010. ISBN 1933988177, 9781933988177.

[91] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

[92] D. A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 235–246. ACM, 1984.

[93] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[94] S. Nettles, J. O'Toole, D. Pierce, and N. Haines. *Replication-based incremental copying collection*. Springer, 1992.

[95] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ACM Sigplan Notices*, volume 50, pages 675–690. ACM, 2015.

[96] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation, OSDI'16, pages 349–365, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026905.

[97] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

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

[99] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), Aug. 2007. ISSN 0164-0925. doi: 10.1145/1255450.1255453. URL http://doi.acm.org/10.1145/1255450.1255453.

[100] F. Petrini and W.-c. Feng. Improved resource utilization with buffered coscheduling. *PARALLEL ALGORITHMS AND APPLICATION*, 16(2):123–144, 2001.

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

[102] D. Porto, J. Leitão, F. Junqueira, and R. Rodrigues. The tortoise and the hare: Characterizing synchrony in distributed environments. In *Proceedings of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

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

[104] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.

[105] S. Salihoglu and J. Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

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

[107] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[108] S. Soman, C. Krintz, and L. Daynès. Mtm2: Scalable memory management for multi-tasking managed runtime environments. In *ECOOP 2008–Object-Oriented Programming*, pages 335–361. Springer, 2008.

[109] 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.

[110] C. J. Stephenson. New methods for dynamic storage allocation (fast fits). In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 30–32, New York, NY, USA, 1983. ACM. ISBN 0-89791-115-6. doi: 10.1145/800217.806613. URL `http://doi.acm.org/10.1145/800217.806613`.

[111] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 international conference on Management of data*, pages 1125–1134. ACM, 2013.

[112] A. S. Tanenbaum. *Modern operating systems*. Prentice Hall Press, 2007.

[113] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46(11):79–88, 2011.

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

[115] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[116] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan Notices*, 19(5):157–167, 1984.

[117] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN Notices*, volume 23, pages 1–17. ACM, 1988.

[118] R. Van Bruggen. *Learning Neo4j*. Packt Publishing Ltd, 2014.

[119] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[120] 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.

[121] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002. ISSN 0163-5980. doi: 10.1145/844128.844146. URL http://doi.acm.org/10.1145/844128.844146.

[122] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 27–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2100-6. doi: 10.1145/2464157.2466481. URL http://doi.acm.org/10.1145/2464157.2466481.

[123] T. White. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

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

[125] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 61–72, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029881. URL http://doi.acm.org/10.1145/1029873.1029881.

[126] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 103–116, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298466.

[127] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[128] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[129] 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.

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

[131] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 21–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508297. URL http://doi.acm.org/10.1145/1508293.1508297.

[132] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. 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, pages 177–188, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024415. URL http://doi.acm.org/10.1145/1024393.1024415.