Analysis of Garbage Collection Patterns to Extend Microbenchmarks for Big Data Workloads

Samyak S. Sarnayak*
samyakssarnayak@pesu.pes.edu
PES University
Bangalore, Karnataka, India

Aditi Ahuja†
aditiahuja@pesu.pes.edu
PES University
Bangalore, Karnataka, India

Pranav Kesavarapu‡
pranavkesavarapu@gmail.com
PES University
Bangalore, Karnataka, India

Aayush Naik§
ayush@pesu.pes.edu
PES University
Bangalore, Karnataka, India

Santhosh Kumar V.¶
santhoskumarv@pes.edu
PES University
Bangalore, Karnataka, India

Subramaniam Kalambur‖
subramaniamkv@pes.edu
PES University
Bangalore, Karnataka, India

ABSTRACT

Java uses automatic memory allocation where the user does not have to explicitly free used memory. This is done by the garbage collector. Garbage Collection (GC) can take up a significant amount of time, especially in Big Data applications running large workloads where garbage collection can take up to 50 percent of the application's run time. Although benchmarks have been designed to trace garbage collection events, these are not specifically suited for Big Data workloads, due to their unique memory usage patterns. We have developed a free and open source pipeline to extract and analyze object-level details from any Java program including benchmarks and Big Data applications such as Hadoop. The data contains information such as lifetime, class and allocation site of every object allocated by the program. Through the analysis of this data, we propose a small set of benchmarks designed to emulate some of the patterns observed in Big Data applications. These benchmarks also allow us to experiment and compare some Java programming patterns.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Runtime environments*; **Garbage collection**; • **General and reference** → **Performance**.

KEYWORDS

Java, Java Virtual Machine, Garbage Collection, Hadoop, Big Data

ACM Reference Format:
Samyak S. Sarnayak, Aditi Ahuja, Pranav Kesavarapu, Aayush Naik, Santhosh Kumar V., and Subramaniam Kalambur. 2022. Analysis of Garbage Collection Patterns to Extend Microbenchmarks for Big Data Workloads. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491204.3527473

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04…$15.00
https://doi.org/10.1145/3491204.3527473

1 INTRODUCTION

Big Data is a phrase that is commonly used to describe the collecting, processing, analysis, and visualisation of large data collections [6, 7]. In recent years, Big data applications have been dominated by Java. Its short development cycles and community support have played a significant role in this. The Garbage Collectors are an important component of the Java Virtual Machine [4, 23]. As garbage collectors can take up a significant amount of time, micro-benchmarks are an important tool to quantify and compare the impact garbage collection has on the performance and efficiency [9, 20]. This is especially true in Big Data applications running large workloads where garbage collection can take a non-trivial percentage of the application's running time [12].

Although benchmarks have been designed to trace garbage collection events [8], these are not specifically suited for Big Data workloads which have garbage collection patterns that are different from most other Java programs.

Detailed and fine-grained data is required to perform an in-depth study of garbage collection patterns and compare the patterns resulting from GC benchmarks and Big Data applications. There exist many profiling tools for Java that provide actionable insights into the Java heap space and objects that take up memory space, such as VisualVM [10]. The level of information provided by these tools is not adequate to perform *analysis on the lifetime of objects*. For example, to figure out if one class of object allocated in a certain function lives longer than the same object allocated elsewhere. This type of analysis requires individual object data with lifetime, class and allocation site details. AntTracks provides a JVM and a visualising tool which can extract data at an individual object level. We extend the AntTracks JVM and build a pipeline around it to provide object-level data and insights. The object analyzer pipeline is open sourced and the tool has been packaged in an easy to use docker image [1].

The main contributions of this work are:

(1) Developing a methodology and improving existing tools to provide per object statistics in an efficient manner.

[1] https://github.com/metonymic-smokey/JavaGC. This repository also contains the code for the newly developed benchmarks.

(2) Analysis of object lifetimes and building micro-benchmarks representing Hadoop computations, based on this analysis.
(3) Performance analysis of the garbage collection patterns of these micro-benchmarks to recommend best practices.

2 RELATED WORK

There exist several profiling tools for Java applications viz. VisualVM and NetBeans profiler. It should be noted that although these tools provide aggregate object data such as the number of objects alive of each type (class) at a given point in time on the heap, it does not include data such as the allocation site, allocation time and lifetime of each object.

AntTracks. Lengauer et al. introduced AntTracks [13–15], a modified Java 8 virtual machine that tracks objects throughout the lifecycle of a Java application. It tracks the birth, death (collected by the GC) and importantly, the movement of objects through the heap. Though heap dumps provide data of objects that are present on the heap at a given point in time, they do not contain the origin (allocation site and time) and deallocation details in them. The JVM does not keep track of unique object identifiers.

Lengauer et al. have changed the interpreter, just-in-time compiler and garbage collector to consider every object allocation and move as an event. Each event is written to the trace file for further analysis. Since there can be millions or even billions of object allocated throughout an application, this object data must be stored in a highly compressed manner. Frequent events are represented more compactly than less frequent occurrences in the custom trace format which is a compact binary file. Some of the event data which can be pre-computed is saved to the trace file. Information that can be rebuilt offline without the running JVM is excluded.

Object Data Representation in AntTracks. When an object is moved, its neighbouring objects are likely to be moved too. Thus, along with the offset address of move, the number of adjacent objects moved is also stored. This information, along with object sizes, provides complete information to reproduce all object movement. The trace files generated by AntTracks JVM are parsed and analyzed by the AntTracks Analyzer [3]. In our work, we build upon and extend these tools to extract object-level information in Big Data systems and analyze them.

Unique Characteristics of Big Data applications. Big Data applications have unique memory patterns compared to most other Java applications. Xu et al. [22] performed an experimental evaluation of garbage collection patterns in Big Data systems. Unique memory usage patterns were found in Big Data applications - *huge objects* and *long-lived shuffled data*. They also have unique computation features - CPU-intensive operations and *iterative computation*. Concurrent garbage collectors such as CMS [17] and G1 reduce pause time as intended, although this results in contention for CPU time with the data operations. All of the three garbage collectors compared (CMS, G1 and Parallel [18]) have been found to be inefficient for large objects with frequent GC cycles observed and in some cases, out of memory errors (in G1 GC). In iterative applications where accumulated data is reclaimed after every iteration, all of the three garbage collectors have been found to be inefficient.

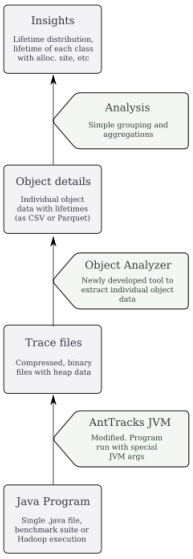

Figure 1: The new object analyzer pipeline. The pipeline extracts object-level data from any Java program using a modified AntTracks JVM and the object analyzer tool. This object data can then be analyzed to extract insights related to object lifetimes and allocation sites.

3 METHODOLOGY

3.1 Object-level data collection pipeline

We present a pipeline to obtain and leverage object-level data, such as lifetime, type and allocation site of each object allocated by the JVM, from Java 8 applications using a modified AntTracks JVM for analysing aspects such as lifetime distribution, variation of lifetime with object type (or class) and variation with allocation site (line of code where the object was allocated).

3.1.1 AntTracks JVM. The first component of our pipeline is the *AntTracks JVM* which, as described earlier, is a modified JVM that traces all objects allocated along with type, allocation site and many other properties. The collected data is stored in a highly compressed manner in three files - `trace`, `symbols` and `class_definitions` - collectively named *trace files* henceforth. The trace file consists of all the object allocations and the Garbage Collector moves that take place during the execution of a Java runtime application. The symbols file consists of encoded allocation information that is mapped to numeric IDs in the events that occur. This reduces the amount of space it would otherwise take up in the trace. The VM uses an efficient format for storing this data that makes use of VM specific knowledge to compress this information.

Our work extends the AntTracks JVM to support multiple JVMs being invoked from the same directory, with support for separate trace files for each JVM. This scenario is common in Big Data applications and benchmark suites.

3.1.2 Object Analyzer. The trace files produced by the AntTracks JVM are in a specialized format that could only be read by the

AntTracks Analyzer. The Analyzer is a GUI tool in the AntTracks ecosystem that parses trace files and produces various analysis from them. The analysis obtained from the Analyzer was not sufficient for our purposes since it did not provide individual object lifetime data.

To extract the data that was required, a *command-line application (Object Analyzer)* was created using the codebase of AntTracks Analyzer. This command-line tool automated parsing the trace files and extracting object data from it, as an alternative to the GUI. AntTracks allows access to the entire state of the heap at various *events*, which can then be used to find object details. AntTracks emits various types of events; two of them are used for our purposes - *GC start* and *GC end*. This provides us with the heap state before and after every garbage collection event. The heap state includes all the objects along with details such as its class, GC ID (incremental integer) at which it was born, GC ID at which it was last moved (by the GC), allocation site and many others. With this we can track the lifecycle of each object as shown in Table 1.

Unchanged objects are ignored. Whenever the birth of an object is identified, we attach an *atomically increasing integer tag* to the object. This allows us to uniquely identify an object even when it's moved by GC. Since these events only provide the heap state at points of time, the birth and death of every object needs to be associated and the lifetime needs to be calculated from it. Whenever an object born event is triggered, we store the object tag and the GC time to allow constant time lookup. This data is used when the same object is collected to calculate the lifetime of the object using the difference of the current time and the value stored in the data. All of the object's data is then serialized to a CSV (comma separated value) file.

3.1.3 Analysis. The object lifetime data obtained from the previously described Object Analyzer is in the form of a CSV file.

The object details from the larger analysis files (>10GB) did not fit into memory. Considering that constraint, the following were the tools used in the analysis:

- Used the *Dask framework* [5, 19]. Dask allowed us to parallelize data loading and computation. Data is loaded in chunks rather than all at once, allowing computation of files larger than system memory while also parallelizing vectorizable operations.
- Used *Apache Parquet format* [21] files partitioned by type before performing analysis using Dask. As an example, a 16GB CSV was compressed to a 1.3GB parquet file. This led to reduced disk usage, faster and parallelizable data loading and faster computations.

The following analysis were obtained using the analysis script:

(1) *Lifetime distribution:* An initial analysis conducted on the object data is to graph the distribution of object lifetimes. The distribution is visualized using a Kernel Density Estimation (KDE) plot with a Gaussian kernel. This was performed using the Seaborn plotting library. The density plot provides an estimate of how many objects have lived for a particular lifetime. The distribution varies significantly between types of workloads.
(2) *Aggregate lifetimes of each class of object*: The object lifetime data contains the class (or type) of the object. This allows grouping by the class and finding insights into how the lifetime of each class differs. Some classes of objects may live longer than others for various reasons: *1)* A class of objects may be used for global configuration objects. In this case, the objects will be garbage collected only at the end of the program leading to high mean and maximum lifetimes, *2)* A class of objects may be declared in the body of a frequently executed loop. In this case, the objects will have a relatively short lifetime since they will most likely be collected in the next GC cycle. The mean, minimum and maximum lifetimes of each class is calculated and plotted in a bar graph. Only classes which have more than 1000 objects are considered for the analysis.
(3) *Aggregate lifetimes of each class with allocation site:* Along with the class, the allocation site of every object is also captured. The allocation site of an object is the function and line number in the source code at which the object was allocated. With this data we can differentiate an object allocated in a hot function and one allocated during configuration. The mean, minimum and maximum lifetimes of each class with allocation site is calculated and plotted in a bar graph. Only classes with allocation sites which have more than 1000 objects are considered for the analysis.

3.2 Analysis of object-level data

The Hadoop examples, described in Table 2, provided a good starting point for garbage collection patterns in Big Data workloads.

Following are some of the observations made using the lifetimes obtained in Figures 2, 3, 4, 5, 6 and 7:

- Map related classes are extensively used in Mapper. `HashMap` related classes have moderate to large mean lifetimes, possibly due to frequent replacement of entries in hash maps, making similar older objects garbage. Most objects live for long enough so they are not garbage collected quickly (in a minor GC).
- For each `HashMap` and `ConcurrentHashMap` entry, a `HashMap $Node` entry is created too. `Node<K,V>` is the inner class used. This can lead to consuming a significant amount of heap space.
- Significant amount of dynamically allocated lists/collections are used in Dancing Links (sudoku) and Terasort. Their underlying `Object[]` arrays are immutable and hence, adding or removing from them allocates more arrays, filling up the heap in a shorter time.
- Proxy generator related classes in Bailey-Borwein-Plouffe (BBP) (Pi) have high mean lifetimes, possibly since the arrays generated internally are immutable.
- Apache unmodifiable entry set is a map that cannot be altered. It has a large mean lifetime since it is a map wrapper which has strong references, hence will be collected after the other maps.

Object State	Description	Event	Identification
Born	object was allocated by the JVM	GC Start	Current GC ID == Object's birth GC ID
Moved	object was moved to a different address during GC (usually due to compaction)	GC End	Last GC ID == Object's last moved GC ID
Death	object was collected or destroyed by the GC	GC end	Object lies in space being collected and was not moved
Unchanged	object was not born, moved or collected	Any	None

Table 1: Object lifecycle tracking and identification of states.

Name	Description	Input type and size
Pi	Estimates value of Pi using the quasi Monte Carlo method	-
Pi (BBP)	Computes exact digits of Pi using Bailey-Borwein-Plouffe algorithm [2]	100000 digits and 16 maps
Sudoku	Solves a Sudoku puzzle using the Dancing Links algorithm [11]	-
Word Count	Counts the number of occurences of each word in a text file	100MB text file
Terasort	Generates and sorts a large, pseudo-random file	1GB binary file

Table 2: Hadoop example programs and their specifications

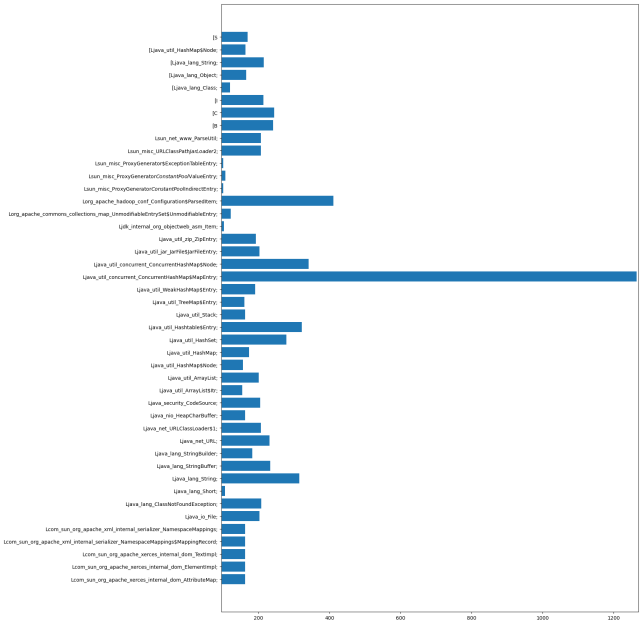

Figure 2: Mean lifetimes (in milliseconds) of classes as seen in an execution of the Pi program. It can be observed that HashMap entries have a relatively larger lifetime, with ConcurrentHashMap entries having the highest lifetimes. When the data is additionally grouped by allocation site and plotted (not shown here), it provides content to this lifetime data. For example, it can be seen that HashMap entries are allocated in the EntryIterator. This means that while iterating over HashMaps in this program, the entries are held in memory for a long time.

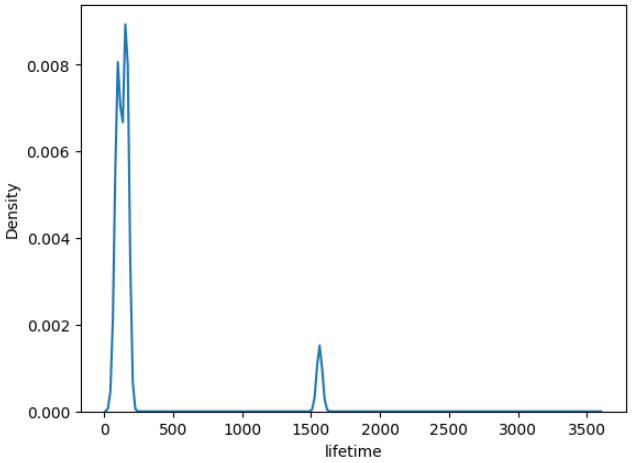

Figure 3: Distribution of lifetimes (in milliseconds) in the Pi program. Most objects live for less than a quarter of a second. There is a small peak at around 1.5s and there are a very small number of objects that live for 3.5s.

3.3 Custom benchmarks

Based on the access patterns observed in the Hadoop results, we codified some of the patterns leading to custom benchmarks implemented with JMH, having the Ionut benchmarks [8] as a reference. The benchmarks tested the following patterns:

- *dynamic vs fixed size array allocation* - This was added since a significant amount of dynamically allocated lists/collections were being used in Dancing Links (Sudoku) [1].
- *using primitive vs boxed type integers.*
- *string concatenation using implicit string addition vs using StringBuilder* - This was considered since a significant amount of implicit string addition is taking place and toString() is invoked, in the Dancing Links (Sudoku) programs.

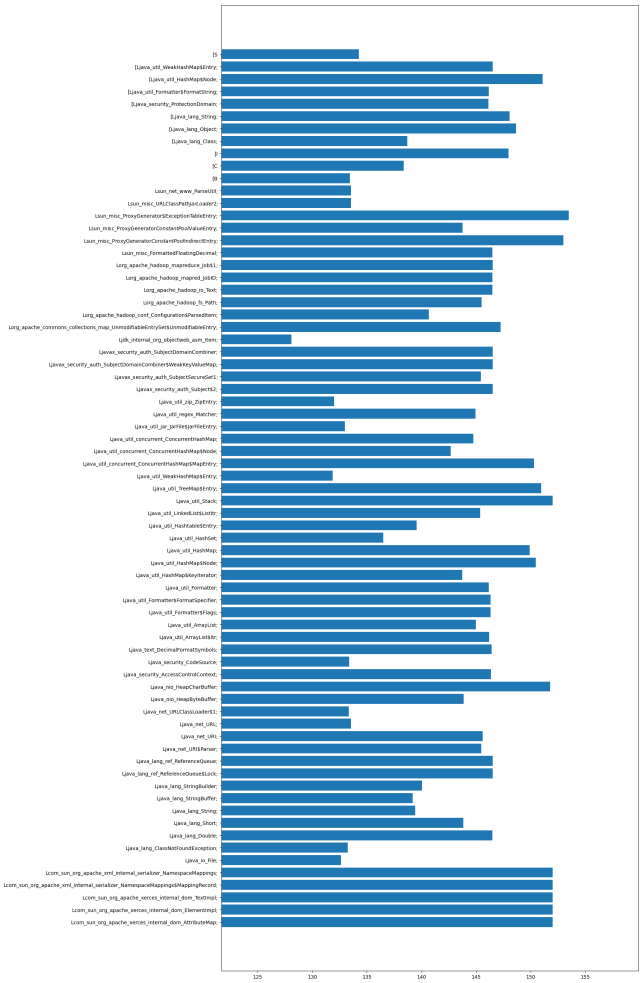

Figure 4: Mean lifetimes (in milliseconds) of classes as seen in an execution of the Pi (BBP) program. UnmodifiableEntrySet and ProxyGenerator related classes can be observed to have large lifetimes.

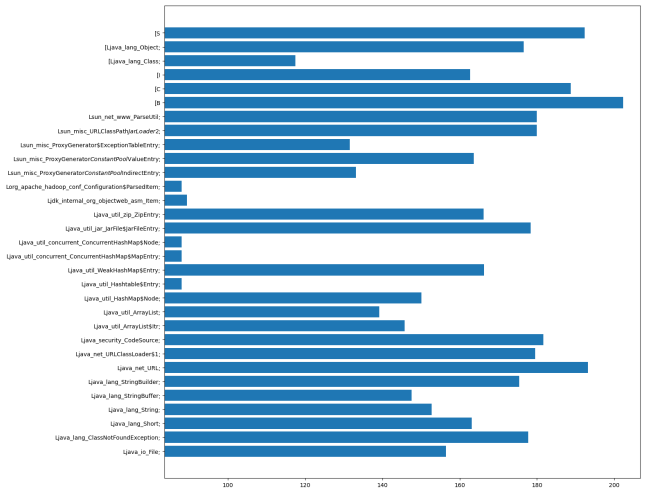

Figure 5: Mean lifetimes (in milliseconds) of classes as seen in an execution of the Sudoku program. Large lifetimes can be observed for arrays, specifically byte arrays (which are represented by `[B`).

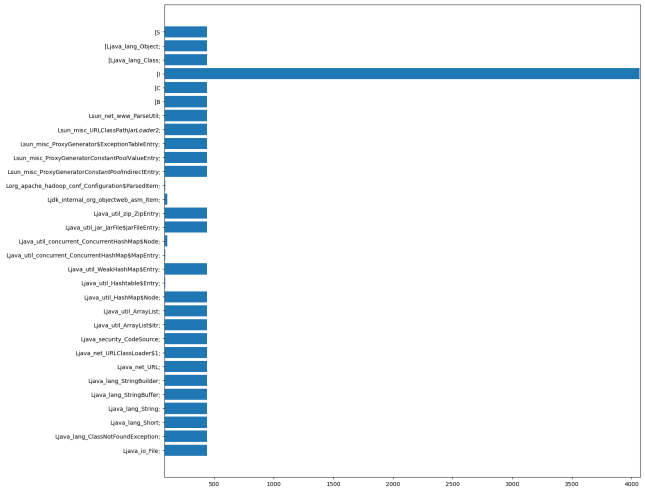

Figure 6: Maximum lifetimes (in milliseconds) of classes as seen in an execution of the Sudoku program. Integer arrays have significantly higher lifetimes since they are used internally by dynamic arrays, such as ArrayList.

4 RESULTS AND ANALYSIS

4.1 Experimental Setup

The Hadoop examples and custom benchmarks were run on a virtual machine with the following specifications:

(1) CPU: AMD Ryzen 7 2700X. 8 cores (vCPUs), 3.7GHz clock frequency, 256 KiB L1 cache, 4 MiB L2 cache, 64 MiB L3 cache, 64-bit.
(2) Memory: 16GB DDR4.
(3) Storage: 400GB SSD.

The following software versions were used:

(1) Operating System: Ubuntu 20.04.3 LTS Focal Fossa.
(2) Container runtime (for Hadoop): Docker 20.10.10 Community.
(3) Hadoop 3.3.1.
(4) Java 8 (AntTracks JVM modified from `jdk8u202-b00`).
(5) JMH 1.27.

4.2 Custom Benchmarks

These micro-benchmarks were developed to measure the impact of garbage collection of common data structures like arrays and strings on the performance of Hadoop workloads. We then suggest some best practices based on the observations and reason out possible causes for the observed impact.

They were implemented using the Java Micro-bench Harness (JMH) framework [16]. The benchmarks were run with 5 warm-up

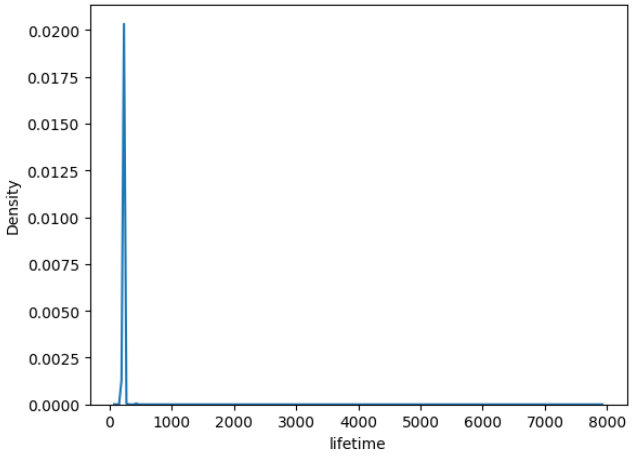

Figure 7: Distribution of lifetimes (in milliseconds) in the Terasort program. The majority of objects live for less than a third of a second. Although, there are some objects that live up to 8 seconds.

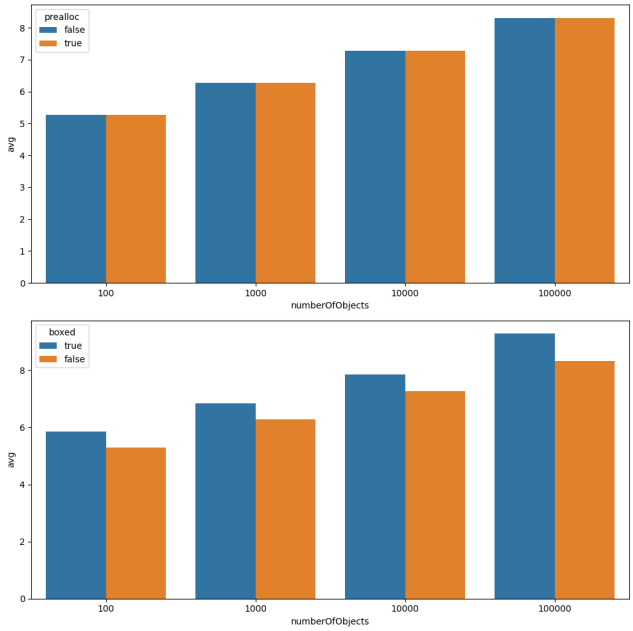

Figure 8: Comparison of execution times (log_{10} of milliseconds) of the Array Benchmark with varying number of objects and with pre-allocation and boxing switched.

iterations and 5 measurement iterations. The average time taken to complete each iteration is used as the metric [2]. The insights obtained from them are described in this section.

4.2.1 Array Benchmark. This benchmark compares the effects of pre-allocating an array vs. adding elements to an empty array (dynamically allocated internally) and using boxed elements instead of primitive types.

Pre-allocated vs. dynamically allocated arrays: Byte has a similar mean lifetime in both. Object has a longer mean lifetime in dynamically allocated arrays since it will be referenced each time the capacity of the initial array is exceeded.

Boxed vs. primitive types: Byte in the boxed graph has twice the mean lifetime of the primitive byte, since it is referenced by `java.lang.Byte` which has a longer lifetime. `java.lang.Byte` has an exceedingly long mean lifetime, as expected.

Insights

Running the Array Benchmark on various number of objects with pre-allocation and boxing switched, we obtained the following insights as seen in Figure 8:

- Pre-allocating a large array does not provide any runtime performance benefits.
- Boxing has a large overhead in runtime performance. It is best to avoid Boxed types when not necessary, although the nature of Java generics requires the use of Boxed types in some cases.
- Arrays scale linearly with a number of elements.

4.2.2 Multiple Array Benchmark. This benchmark compares allocating multiple small arrays vs. a single large array.

Splitting a large array into multiple smaller arrays adds the overhead of extra padding and reference size bytes for each smaller array. Large array list lasts longer possibly because it uses less memory than smaller arrays so the heap fills up slower. Object type, on an average, lasts longer in a larger array than a smaller one, possibly since instantiating more objects takes more bytecode instructions.

Insights

Running the Multiple Array Benchmark on a range of number of small arrays and a range of bytes provided the following insights as seen in Figure 9:

- Using one large array or a small number of large arrays is preferable to having hundreds or thousands of small arrays. This is due to the object reference overhead for each array. Having multiple small arrays leads to fragmentation in memory.
- Performance degrades quickly when a very large number of smaller arrays are used. When thousands of arrays are present, most of the computation time is used in the overhead of the internals of these arrays.

4.2.3 String Benchmark. This benchmark compares the garbage collection resulting from string concatenation using implicit string additions vs. using `StringBuilder()`.

- There is a very minor difference in the mean lifetime of objects in both cases.
- The String object when using `StringBuilder()` has a minimum and maximum lifetime greater than when using implicit addition.
  This could be possibly attributed to the fact that using implicit additions creates extra objects, which fill up the heap

[2]The complete data including min, max, standard deviation and 99.9% CI is provided in the same code repository.

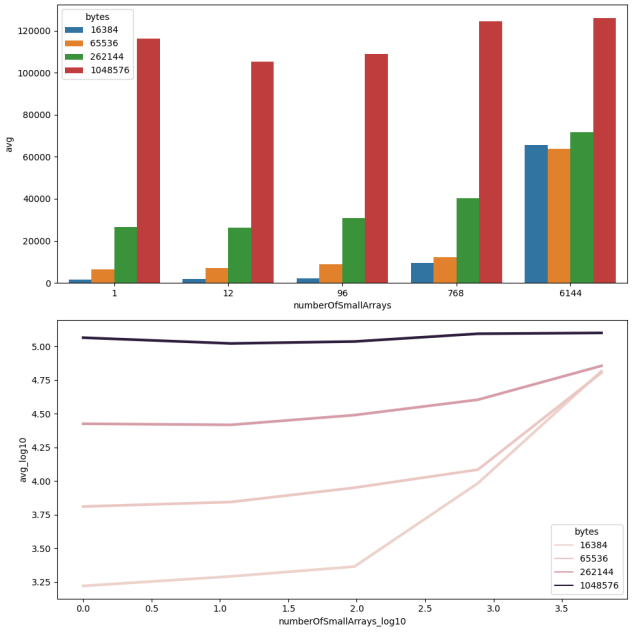

Figure 9: Comparison of execution times of the Multiple Array Benchmark with varying number of small arrays and number of bytes. Note that the first graph uses a linear scale whereas the second uses a log10 scale.

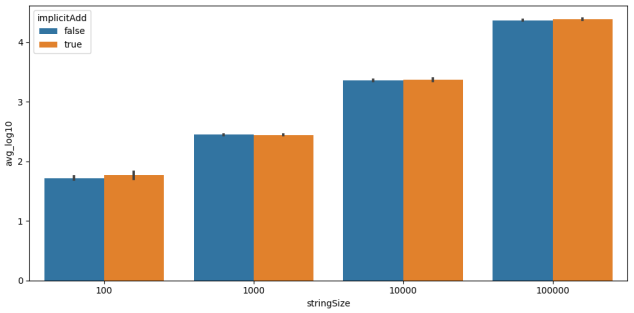

Figure 10: Comparison of execution times (log_{10} of milliseconds) of the String Benchmark with implicit addition and `StringBuilder()` switched.

faster, triggering GC and hence, the lifetime of the objects is shorter.

Insights

Running the String Benchmark on a range of string sizes and implicit addition/`StringBuilder()` switched the following insights as seen in Figure 10:

- There is a very negligible difference in the runtime performance of using `StringBuilder()` and using implicit string addition. Although `StringBuilder()` can be slightly more performant, the convenient syntax of implicit addition makes it the clear choice when it is applicable.
- String addition scales linearly with the string sizes.

5 CONCLUSIONS AND FUTURE WORK

We developed a pipeline to extract and analyse object level patterns in any Java program and obtained data for Hadoop workloads since Big Data workloads have object lifetime patterns different from those of standard benchmarks. Using the data obtained, we developed a set of benchmarks to mimic some patterns noticed in Big Data applications, and to understand the effects of using different Java constructs or data structures.

The object data pipeline includes a modified AntTracks JVM, a newly developed Object Analyzer, data conversion (CSV to parquet) and analysis scripts (implemented using Dask). This pipeline is the result of experimenting with and comparing various tools. We analyzed garbage collection patterns in Big Data workloads and GC benchmarks using the pipeline. We then implemented analysis scripts on the obtained CSV files in an optimized manner to support hundreds of millions of objects.

The insights obtained from the custom benchmarks show that adding elements to an array and string addition both scale linearly with size. Pre-allocating an array does not provide any performance benefit whereas using a primitive type instead of a boxed one as array elements does. Using `StringBuilder()` in cases where string addition can be used does not provide any performance benefit. From the multiple array benchmark, it can be concluded that a large number of very small arrays should be avoided and a small number of (or one) large arrays should be used instead.

Our future work will involve further analysis on a variety of Big Data workloads, with a focus on the internals of common Big Data tools since they play the greater role in most such workloads. Our future work will also involve running these on multiple nodes since that will give us a more realistic idea of how these workloads perform in real-life deployments, which are usually distributed and rarely single-node.

ACKNOWLEDGMENTS

We would like to express our gratitude to Markus Weninger, one of the maintainers of AntTracks for updating the AntTracks source code, guiding us through the AntTracks Analyzer source code and his prompt responses to our queries.

REFERENCES

[1] Apache Hadoop team. 2012. *Dancing Links codebase.* http://svn.apache.org/viewvc/hadoop/common/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/dancing/

[2] David H. Bailey. 2006. The BBP Algorithm for Pi. *Lawrence Berkeley National Laboratory* (2006).

[3] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (Melbourne, FL, USA) *(PPPJ '15)*. Association for Computing Machinery, New York, NY, USA, 76–89. https://doi.org/10.1145/2807426.2807433

[4] Yu Chan, Andy Wellings, Ian Gray, and Neil Audsley. 2014. On the Locality of Java 8 Streams in Real-Time Big Data Applications. In *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems* (Niagara Falls, NY, USA) *(JTRES '14)*. Association for Computing Machinery, New York, NY, USA, 20–28. https://doi.org/10.1145/2661020.2661028

[5] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org

[6] Isitor Emmanuel and Clare Stanier. 2016. Defining Big Data. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies* (Blagoevgrad, Bulgaria) *(BDAW '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3010089.3010090

[7] Palak Gupta and Nidhi Tyagi. 2015. An approach towards big data — A review. In *International Conference on Computing, Communication Automation*. 118–123. https://doi.org/10.1109/CCAA.2015.7148356

[8] Ionut Balosin. 2019. *Ionut Benchmarks*. https://ionutbalosin.com/2019/12/jvm-garbage-collectors-benchmarks-report-19-12/

[9] Nusrat Sharmin Islam, Xiaoyi Lu, Md. Wasi-ur Rahman, Jithin Jose, and Dhabaleswar K. (DK) Panda. 2014. A Micro-benchmark Suite for Evaluating HDFS Operations on Modern Clusters. In *Specifying Big Data Benchmarks*, Tilmann Rabl, Meikel Poess, Chaitanya Baru, and Hans-Arno Jacobsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–147. https://doi.org/10.1007/978-3-642-53974-9_12

[10] Jiri Sedlacek,Tomas Hurka. 2019. *VisualVM*. https://visualvm.github.io/index.html

[11] Donald E. Knuth. 2000. Dancing links. arXiv:cs/0011047 [cs.DS]

[12] Martin Larose and Marc Feeley. 1998. A Compacting Incremental Collector and Its Performance in a Production Quality Compiler. *SIGPLAN Not.* 34, 3 (oct 1998), 1–9. https://doi.org/10.1145/301589.286861

[13] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/2972206.2972220

[14] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/2668930.2688037

[15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/2851553.2851555

[16] OpenJDK Authors. 2020. *Java Micro-Benchmarks Harness (JMH)*. https://openjdk.java.net/projects/code-tools/jmh/

[17] Oracle. 2019. *Concurrent Mark Sweep (CMS) Collector*. https://docs.oracle.com/en/java/javase/12/gctuning/concurrent-mark-sweep-cms-collector.html

[18] Oracle. 2019. *Parallel Collector*. https://docs.oracle.com/en/java/javase/12/gctuning/parallel-collector1.html

[19] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[20] Sanaz Tavakolisomeh. 2020. Selecting a JVM Garbage Collector for Big Data and Cloud Services. In *Proceedings of the 21st International Middleware Conference Doctoral Symposium* (Delft, Netherlands) *(Middleware'20 Doctoral Symposium)*. Association for Computing Machinery, New York, NY, USA, 22–25. https://doi.org/10.1145/3429351.3431745

[21] The Apache Parquet team. 2013. *Apache Parquet*. https://parquet.apache.org/

[22] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An Experimental Evaluation of Garbage Collectors on Big Data Applications. *Proc. VLDB Endow.* 12, 5 (jan 2019), 570–583. https://doi.org/10.14778/3303753.3303762

[23] Xiaolan Zhang and Margo I Seltzer. 2001. HBench: JGC-An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance.. In *COOTS*, Vol. 1. 4–4.