ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

Java Garbage Collector Selection Using Object Oriented Software Metrics

M.Sc. THESIS

Gencay SAZAK

Department of Computer Engineering

Computer Engineering Programme

DECEMBER 2017

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

Java Garbage Collector Selection Using Object Oriented Software Metrics

M.Sc. THESIS

Gencay SAZAK
(504121519)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Asst. Prof. Dr. Tolga OVATMAN

DECEMBER 2017

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

Nesne Yönelimli Yazılım Ölçütleri ile Java Atık Bellek Toplayıcı Seçimi

YÜKSEK LİSANS TEZİ

Gencay SAZAK
(504121519)

Bilgisayar Mühendisliği Ana Bilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Yrd. Doç. Dr. Tolga OVATMAN

ARALIK 2017

Gencay SAZAK, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 504121519 successfully defended the thesis entitled “Java Garbage Collector Selection Using Object Oriented Software Metrics”, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Asst. Prof. Dr. Tolga OVATMAN** Istanbul Technical University	.....
Jury Members :	**Asst. Prof. Dr. Yusuf Yaslan** Istanbul Technical University	.....
	Asst. Prof. Dr. Amaç Güvensan Yıldız Technical University	.....

Date of Submission : **16 November 2017**
Date of Defense : **22 December 2017**

To my family,

FOREWORD

I would like to express my gratitude to my supervisor Asst. Prof. Dr. Tolga Ovatman. With his guidance and encouragement, I was able to finish my thesis.

I dedicate this thesis to my dear family. I am grateful for everything they have done for me.

Finally, I want to thank all of my friends. They accompanied and supported me always.

December 2017

Gencay SAZAK
Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

AVCC	**:** Average Cyclomatic Complexity
CBO	**:** Coupling between object classes
CMS GC	**:** The Concurrent Mark Sweep Garbage Collector
COH	**:** Cohesion
DIT	**:** Depth of Inheritance Tree
EXT	**:** External Methods
FIN	**:** Fan In
FOUT	**:** Fan Out
INST	**:** Instance Variables
LCOM	**:** Lack of Cohesion in Methods
LMC	**:** Local Methods Called
MAXCC	**:** Maximum Cyclomatic Complexity
MOD	**:** Modifiers
MPC	**:** Message Passing Coupling
NCO	**:** Number of Commands
NOS	**:** Number of Statements
NQU	**:** Number of Queries
PACK	**:** Packages imported
RFC	**:** Response For a Class
RR	**:** Reuse Ratio
SR	**:** Specialization Ratio
TCC	**:** Total Cyclomatic Complexity
UWCS	**:** Unweighted Class Size
G1 GC	**:** The Garbage First Garbage Collector
GC	**:** Garbage Collector
Par GC	**:** The Parallel Garbage Collector
ParOld GC	**:** The Parallel Old Garbage Collector
Ser GC	**:** The Serial Garbage Collector
JDK	**:** Java Development Kit
JVM	**:** Java Virtual Machine

SYMBOLS

GB **:** Gigabyte

MB **:** Megabyte

LIST OF TABLES

Page

LIST OF FIGURES

Page

JAVA GARBAGE COLLECTOR SELECTION USING OBJECT ORIENTED SOFTWARE METRICS

SUMMARY

Java programming language introduces garbage collection to manage memory automatically like many other high-level programming languages. Garbage collector both facilitates development and maintenance.

Despite its advantages, garbage collector uses system resources and it imposes overhead on programs. Two types of these overhead are particularly important: First, garbage collector affects total execution time of program. Secondly, garbage collector stops program while working making it unresponsive. Therefore garbage collection efficiency is important and program performance can be enhanced by using good performing garbage collector.

Every garbage collector has its own pros and cons. There is no garbage collector that outperforms others in all cases. Which garbage collector is the best for the program dependes on the code. Therefore in this thesis, it is proposed that there should be a relation between garbage collector performance and program properties. It is also proposed that this relation can be exploited and good performing garbage collector for a program can be predicted by analyzing program.

Predictor function can be constructed by using a machine learning system which creates a decision tree. Training data is needed to feed such system. First step to obtain this data is to find sample programs which have distinct characteristics. These programs should reflect real world and also stress system memory to produce valuable garbage collection information. These programs can be tested with different garbage collectors. If different garbage collectors perform best for different programs, then relation between garbage collection performance and program properties is proven. After that program property data can be merged to garbage collector performance data to produce training data. With this training data a decision tree can be produced. Decision tree can predict garbage collector performance and show which properties are related to garbage collection performance.

OpenJDK is chosen for the work. Because OpenJDK is both popular in industry and academy. OpenJDK 8 is chosen as version as it is both mature and recent. OpenJDK contains 5 different garbage collectors. These collectors are Serial Garbage Collector, Parallel Garbage Collector, Parallel Old Garbage Collector, Concurrent Mark Sweep Collector and Garbage First Garbage Collector.

Two criteria are chosen to define garbage collector performance. These are total execution time of program and total pause time of program due to garbage collection. Lower values are better. For each garbage collector, total execution time and total pause time are compared to best values. If both values are less than or equal to 120% of best values, garbage collector is considered acceptable for given program.

The DaCapo Benchmark Suite is chosen for work. DaCapo contains many programs which satisfy requirements for work. It also facilitates experiments and analysis as it is open source.

A benchmark driver program is implemented to run DaCapo programs with different configurations in separate processes.

Parallel and Parallel Old garbage collectors performed good in most configurations. Concurrent Mark Sweep garbage collector performed bad in all cases. Remaining collectors performed good in several cases. Results are reported in work.

Benchmark programs are analyzed both statically and dynamically. Metrics including Chidamber and Kemerer metrics are extracted as static properties and also object allocation metrics extracted as dynamic properties through a single program execution. Program properties mentioned constructed training data with garbage collector results. Program properties acquired reported in the work.

After obtaining training data, Weka machine learning tool is used to build a decision tree. J48, which is an implementation of C4.5 algorithm is used to build a decision tree. Decision tree's success rate is measured as 76.85%.

With this work, garbage collection performance and program properties of DaCapo benchmarks are reported using a recent system configuration. This report may be useful in future research on DaCapo. The relation between program properties and garbage collection performance is shown. Using this relation a predictor system is proposed. This system had been able to classify instances correctly. In future research, it is possible to obtain better results with extended set of sample programs and program properties.

NESNE YÖNELİMLİ YAZILIM ÖLÇÜTLERİ İLE JAVA ATIK BELLEK TOPLAYICI SEÇİMİ

ÖZET

Birçok yüksek düzeyli programlama dilinde olduğu gibi Java programlama dilinde de otomatik bellek yönetimi yapılmaktadır. Bellek yönetimi, atık bellek toplayıcı tarafından yapılmakta ve program tarafından kullanılmayan nesneler bellekten kaldırılmaktadır. Atık bellek toplayıcı, bu özelliğiyle hem yazılımcının geliştirmesini kolaylaştırmakta, hem de kodun bakım yapılabilirliğini artırmaktadır.

Atık bellek toplayıcı, yazılım geliştirmeye getirdiği avantajlarla beraber, çalışma esnasında sisteme ek yük oluşturmaktadır. Özellikle iki çeşit yük dikkat çekmektedir: Atık bellek toplayıcı, belleği yönetirken sistem kaynaklarını kullandığı için programın toplam çalışma süresini de etkilemektedir. Ayrıca, toplama işlemi sırasında programın çalışmasını da duraksatmaktadır. Bu duraksamalar, programın cevap verebilirliğini olumsuz yönde etkilemektedir. Bu sebeplerden ötürü, atık bellek toplayıcının verimli çalışması oldukça önemlidir. Mevcut atık bellek toplayıcılardan iyi performans sunanları seçmek de verimi artıracaktır.

Her atık bellek toplayıcının güçlü ve zayıf yönleri vardır. Dolayısıyla bütün senaryolarda en iyi performansı veren bir atık bellek toplayıcı mevcut değildir. Bazı programlarda en iyi performansı veren atık bellek toplayıcı, bir başka programda aynı performansı vermeyebilir. Bu çalışmada, program özellikleri ile atık bellek toplayıcı performansı arasında bir ilişki olduğu iddia edilmektedir. Bu ilişkinin tespit edilerek, program analizi ile bu programda iyi performans verecek atık bellek toplayıcının tahmin edilebileceği düşünülmektedir.

Tahmin fonksiyonu, karar ağacı öğrenen makine öğrenmesi sistemleri ile oluşturulabilir. Sistemi beslemek içinse öncelikle eğitim verisinin oluşturulması gerekmektedir. O halde, öncelikle farklı karakteristiklere sahip örnek programlar bulunmalıdır. Bu programlar, gerçekte kullanılan programları yansıtmalı ve sistem belleği üzerinde çok yük oluşturarak atık bellek toplama verisinin anlamlı olmasını sağlamalıdır. Bulunan programlar, farklı atık bellek toplayıcılar ile test edilerek atık bellek toplayıcıların performansı incelenmelidir. Eğer farklı programlarda farklı atık bellek toplayıcıların iyi performans verdiği tespit edilirse, program özellikleri ile atık bellek toplayıcı performansı arasında ilişki bulunduğu gösterilmiş olur. Ardından, programların özellikleri çıkarılarak, performans verisi ile birleştirilip eğitim verisi oluşturulabilir. Bu veriden oluşturulacak geçerli bir karar ağacı hem iyi atık bellek toplayıcı seçebilmeyi sağlayacak, hem de performansın hangi özelliklere bağlı olduğunu gösterecektir.

Çalışma için OpenJDK sanal makinesi seçilmiştir. OpenJDK, hem endüstride yaygın olarak kullanılan hem de akademik çalışmalarda da tercih edilen bir platformdur. Bu sebeple bu çalışma için de tercih edilmiştir. Versiyon seçiminde ise, hem yeterince yeni hem de kararlı bir sürüm olan OpenJDK 8 seçilmiştir. OpenJDK, 5 farklı atık

bellek toplayıcı içermektedir. Bunlar; Serial, Parallel, Parallel Old, Concurrent Mark Sweep ve Garbage First atık bellek toplayıcılarıdır.

Bir atık bellek toplayıcının performansını değerlendirmek için iki kriter seçilmiştir: Birinci kriter, programın toplam çalışma süresi, ikinci kriter ise programın atık bellek toplama işlemi sebebi ile toplam duraksama süresidir. Bu sürelerin kısa olması gerekmektedir. Bir atık bellek toplayıcının bir programdaki performansının iyi olup olmadığının belirlenmesi için bu atık bellek toplayıcının toplam çalışma süresi ve toplam duraksama süresi, söz konusu programda en iyi performansı veren atık bellek toplayıcının süreleriyle karşılaştırılmıştır. Atık belleyicinin her iki süresi de, elde edilebilen en iyi sürenin %120'sini geçmiyor ise atık belleyici performansı kabul edilebilir olarak değerlendirilmiştir.

Çalışmayı yapmak için DaCapo performans test seti seçilmiştir. DaCapo, bahsedilen özellikleri sağlayan birçok farklı program içermektedir. Ayrıca, açık kaynak kodlu olması sebebiyle yapılması gereken test ve analizleri kolaylaştırmaktadır. Testler yapılırken DaCapo'nun dahili bileşenlerinden faydalanılmıştır. Kararlı sonuçları raporlayabilmek için, dahili ısınma mekanizması kullanılmıştır. Bu mekanizma ile test, birden fazla kere çalıştırılmakta ve sonuçlar birbirine yakınsadığında artık kararlı olduğu tespit edilerek raporlanmaktadır. Kullanılan bir başka özellik, geri çağırma mekanizmasıdır. Bu özellik kullanılarak, test bittiğinde çalışacak bir kod eklemesi yapılarak sonuçların daha sonra işlenmek üzere bir metin dosyasına kaydedilmesi sağlanmıştır.

DaCapo programlarının üzerinde çalışılacak tüm atık bellek toplayıcılar ve bellek boyutları ile test edilmesi gerekmektedir. Bunun için bir sürücü program geliştirilmiştir. Sürücü program, Java sanal makinesi ve DaCapo setinin parametrelerini ayarlayarak tüm bahsedilen tüm kombinasyonları ayrı bir işlemde çalıştırmaktadır.

Test için Microsoft Azure bulut platformunda 4 çekirdek ve 8 gigabayt bellekli sanal makineler oluşturulmuştur. Makinelere işletim sistemi olarak Ubuntu 16.04 sunucu versiyonu ve OpenJDK 8 Java sanal makinesi yüklenmiştir. Sanal makine bellek limiti olarak sırayla 1024, 2048, 3072, 4096 megabayt seçilerek testler çalıştırılmıştır.

Testler çalıştırılarak atık bellek toplayıcı performans sonuçları elde edilmiştir. Sonuçlara göre Parallel ve Parallel Old atık bellek toplayıcıları birçok testte kabul edilebilir sonuç vermiştir. Concurrent Mark Sweep atık bellek toplayıcısı ise hiçbir durumda kabul edilebilir performans sunmamıştır. Geri kalan atık bellek toplayıcılar ise farklı durumlarda iyi sonuçlar vermiştir. Söz konusu sonuçlar çalışmada raporlanmıştır.

Programlar analiz edilerek özellikleri tespit edilmiştir. Sabit ve dinamik özellikler seçilmiştir. Sabit özellikler, programın kaynak kodu analiz edilerek tespit edilebilmektedir. Chidamber ve Kemerer ölçütlerini de içeren birçok ölçüt sabit özellik olarak seçilmiştir. JHawk programı ile DaCapo kaynak kodu analiz edilerek özellikler tespit edilmiştir. Dinamik özellikler, programın çalıştırılması ile elde edilebilecek özelliklerdir. Dinamik özellikler ile programın bellek kullanım karakteristiği tespit edilebilir. Dinamik özellik olarak: Java standart kütüphanesi harici sınıf nesnelerinin oluşturulma oranı, dizi oluşturma oranı, karakter dizisi oluşturma oranı, bağlı liste oluşturma oranı seçilmiştir. YourKit Java Profiler programı kullanılarak tüm test programları birer kere çalıştırılmış ve her çalışmada bellekte oluşturulan bütün nesneler sayılmıştır. Bu sayım ile bahsedilen dinamik özellikler tespit edilmiştir. Hem

sabit hem dinamik özellikler çalışmada raporlanmıştır. Özelliklerin tespiti sonucunda test programlarının farklı karakteristiklere sahip olduğu görülmüştür.

Oluşturulan eğitim verisini işlemek için Weka makine öğrenmesi programı kullanılmıştır. Weka, birçok karar ağacı öğrenme algoritmasını kullanabilmek için gerekli arayüzü sağlamaktadır. Eğitim verisi, bir C4.5 karar ağacı gerçeklemesi olan J48 karar ağacı öğrenmesi ile kullanılmıştır. Karar ağacı üzerinde 10 katlı çapraz doğrulama yapılarak geçerliliği ölçülmüştür. Oluşturulan karar ağacı %76,85 başarılı sınıflandırma oranına sahiptir.

Çalışma sonunda DaCapo programlarının atık bellek toplayıcı performansları ve program özellikleri güncel bir sistem konfigürasyonunda ölçülerek raporlanmıştır. Bu çalışmanın DaCapo üzerine ileride yapılacak çalışmalara yardımcı olacağı düşünülmektedir. Program özellikleri ile atık bellek toplayıcı performansı arasında ilişki olduğu gösterilmiştir. Bu ilişkiyi formüle ederek atık bellek toplayıcı performansını tahmin edecek bir sistem önerilmiştir. Önerilen sistem verilen sonuçları sınıflandırabilmiştir. Gelecek çalışmalarda daha geniş bir program örneklemi ve özellik seti seçilerek başarı düzeyinin artırılabileceği düşünülmektedir.

1. INTRODUCTION

Many programming languages utilize garbage collection to automatically manage memory. Despite others, these languages do not require manual object deallocation from memory. According to Jones et al, garbage collection prevents memory management errors and introduces low coupling to applications, therefore eases code maintenance [1]. However, these benefits come with a cost of garbage collection overhead. This overhead can be summarized as follows:

- Decreased Throughput: Garbage collector uses system resources while working. As application shares these resources with garbage collector, its total execution time prolongs.
- Decreased Responsiveness: Garbage collector needs to pause program while working to manage memory correctly. Program can't respond to any action when paused.

It is desired to reduce garbage collection overhead as much as possible. One possible approach is to choose most efficient garbage collector available which is also primary focus of thesis.

Choosing garbage collector is non-trivial, because every garbage collector has its own pros and cons. A garbage collector which performs best in one program may not perform well in another. Therefore in this thesis it is proposed that a relation between garbage collector performance and code properties exists. Using this relation, a method of choosing a good garbage collector without extensively benchmarking application is shown.

In this study, first a program set to experiment on is introduced. Java programming language is focused as it is one of the most popular programming languages and has available tools and works that facilitate research. Experiments are made with program set and it is shown that no garbage collector outperforms others in all cases.

Afterwards, programs are analyzed and their properties obtained. Using data obtained and a decision tree learner, a decision tree to predict feasibility of garbage collection configuration for a given program is proposed.

The rest of thesis organized as follows: In chapter 2, literature on the topic is reviewed. Their similarities and differences in procedure and results to this work are discussed. In chapter 3, brief information on garbage collection is given. In chapter 4, experimental environment used is described. In chapter 5, results of experiment on garbage collection performance are reported. In chapter 6, several properties to define a program proposed and extract process are described. In chapter 7, data obtained in previous chapters is used to train a decision tree. Afterwards, efficiency of decision tree is reported. Finally in chapter 8, results of the work are summarized. Also conclusions and thoughts are shared. Possible ways to extend this work are discussed.

2. LITERATURE REVIEW

There are many researches related to garbage collection performance, especially in Java. Also there are a few researches which investigate relation between implementation and garbage collection performance. Singer et al showed this relation by selecting best performing garbage collector through application analyze [2]. They extracted Chidamber and Kemerer metrics from programs as properties [3]. With a sample execution, heap usage characteristics are also acquired and used as properties. For each case, best garbage collector is determined and training data which includes garbage collector results and program properties aforementioned obtained. Finally training data is used to construct a decision tree which predicts best performing garbage collector. Predictor has been able to improve performance by predicting better garbage collectors instead of default choice. In the paper, several Java benchmarks which includes DaCapo benchmark suite is used to obtain training data. While DaCapo benchmark suite is used only in this thesis, more recent version of suite was available at the time of research which includes more benchmarks than previous version used by Singer et al. Authors experimented with Jikes RVM[1] which is suitable for research purposes. In this work, OpenJDK is used instead as it has actual usage in real-world applications. As simulating more realistic situations is intended, larger heap sizes are used in this research compared to Singer et al's work. Another difference between two works is measurement to define garbage collector performance. Both garbage collection pause time and total execution time are used while they used total execution time. Furthermore, instead of choosing best collectors as it is in paper by Singer et al, decision tree in this thesis marks given garbage collection configuration as acceptable or unacceptable. Because in many cases, more than one garbage collector yield similar results so generalized form of measurement is used. Aforementioned evaluations are reported using different algorithms and different algorithm parameters.

[1] http://www.jikesrvm.org/

Lengauer and Mössenböck improved garbage collection performance by finding optimal garbage collector parameters for each collector [4]. Given possible parameter values, their optimization function executes program using these parameters and finds best configuration. This approach differs from this work as default configuration for all collectors is used.

Libič et al approached garbage collection performance in a different way by building a function to estimate effect of code changes to collection performance [5]. Introduced function compares collection behaviour difference after code changes by examining gc logs provided by virtual machine. However no static analysis of code is used in this model.

Lengauer et al studied garbage collection and memory features of different benchmarks [6]. Authors included DaCapo benchmark suite along with DaCapo Scala[2] and SPECjvm2008[3] in their research. They shared allocation statistics for each benchmark. In addition, comparison between Parallel Old GC and G1 GC in terms of garbage collection time and garbage collection pause time given. Best garbage collector configurations in their results are similar to thesis' results.

Eran and Petrank studied garbage collection performance of data structures in many-core environments [7]. They found out that data structures like queues and linked lists can't be processed in parallel by garbage collector effectively. Therefore, these data structures reduce scaling of applications. They also introduced alternative data structures to enable efficient parallel garbage collection. Linked list usage percentage is included in thesis to assess mentioned behaviour.

Carpen et al evaluated garbage collector performance in a server-scale environment which has 48 cores and 64GB RAM [8]. They used a real-world Cassandra[4] database application in addition to DaCapo benchmark suite for experiments. They showed that garbage collection affects application performance severely with pause times range from seconds to minutes. Similar to our findings, they concluded that Parallel Old Collector performs well with DaCapo benchmarks having less pause time and

[2] http://www.scalabench.org/
[3] https://www.spec.org/jvm2008/
[4] http://cassandra.apache.org/

execution time. However it should be noted that G1 collector had better pause time when database application is used as benchmark.

Paper by Gidra et al is another research that discusses scaling of garbage collection as core count increases [9]. Findings are similar to aforementioned, garbage collector acts as a bottleneck in parallel environment although they measured collectors that utilize multiple threads. They also showed that scalability differs between different benchmarks.

Detlefs et al introduced G1 collector in their paper [10]. Aside from G1 collector's details of design, performance evaluation is also given. G1 achieved better pause time than other collectors. On the other hand, no improvements observed in terms of throughput. Authors noted that pause time measurement is considered more important than throughput. Also in this thesis, it is observed that G1 had better responsiveness in several cases but never had better throughput than other collectors.

Most of related works evaluate garbage collection performance using DaCapo benchmark suite and OpenJDK VM. Garbage collection performance results of this configuration also provided in thesis. Some of aforementioned works introduce methods to improve garbage collection performance. However none of them offers performance prediction using program analysis except Singer et al's paper. Prediction technique used in this thesis is similar to Singer et al's, however experimental setup and evaluation is changed to be relevant to current usage in industry.

3. GARBAGE COLLECTION BACKGROUND

In Java, a memory region stores all created objects in program. This region is called heap. JVM Specification by Lindholm et al implies that Java is one of the programming languages that use garbage collection [11]. Therefore, heap is processed by garbage collector and unused objects are automatically removed from heap. Several garbage collectors are implemented for this purpose. In this chapter, general features in Java garbage collection will be discussed first. After that, virtual machine implementation and garbage collectors used in our work will be introduced.

3.1 Garbage Collection Features

Garbage collector uses system resources to work. Furthermore, it stops the application while working. It is called stop-the-world-pause and according to Jones et al, this pause is needed to prevent object reference changes which can't be tracked by collector while working [1]. Application can't respond while stopped by collector therefore its responsiveness is affected in addition to its execution time.

All garbage collectors in Java are generational, which means heap is split into regions called generation. Generation where object resides is determined by object's age. Generally, newly created objects are placed into young generation and when they get older they are moved to old generation. Jones et al mention that this behaviour is to utilize the assumption that new objects' expected life-span is short [1].

3.2 Garbage Collection OpenJDK VM

OpenJDK is used as JVM in this thesis, because it is open-source, subject to many academic researches and is also widely used in industry. OpenJDK 8 is chosen as version because it is a recent yet mature release. OpenJDK implements different garbage collectors so it is possible to test same program with different garbage collectors. Implemented garbage collectors are described below.

- Serial Garbage Collector: Ser GC is the most basic collector. It only uses single thread to operate and stops the application while collecting.
- Parallel Garbage Collector: Par GC utilizes multiple cores to work. This results in faster collection thus it is also named as throughput collector. Par GC uses stop-the-world-pause like Ser GC.
- Parallel Old Garbage Collector: ParOld GC is a close variant of Par GC. This collector uses multiple threads for collecting young generation and uses single thread for collecting old generation.
- Concurrent Mark Sweep Collector: CMS GC introduced in JDK 1.4.1 to address stop-the-world-pause problem. In contrast to collectors aforementioned, CMS GC can run concurrently with application. However it still uses application pause while working. These pauses are shorter than serial and parallel collectors though.
- Garbage First Garbage Collector: G1 GC is the most recent collector which is introduced in JDK7 update 4 and implemented by Detlefs et al [10]. It is planned to replace CMS GC in future.

4. EXPERIMENTAL SETUP

This chapter describes collecting process and design choices of training data. Training data is used to feed machine learning system which in turn produces garbage collector selector.

4.1 Garbage Collector Comparison

Before starting data collect, it is needed to define what makes a garbage collector better than rest. Jones et al describe several properties to compare collectors [1]. Among them, two are picked: Throughput, which means total execution time of whole application and GC pause time, total amount of time application stopped by collector. In this work, a soft criterion is used to measure garbage collector performance instead of choosing best garbage collector. For each program run, all garbage collectors' total execution time and GC pause time are compared to best performing garbage collector's. Unless execution time or pause time are more than 120% of best garbage collector's measurement for that program, given garbage collector is considered as acceptable. Otherwise, garbage collector performance is unacceptable and should not be used with given program. This threshold is chosen to cover decent amount of cases as acceptable. Because lower threshold may result in marking most cases as unacceptable, therefore decision tree can't predict performance effectively. On the other hand, if threshold is chosen too high, performance gap between best garbage collector will be high which is not desired. Decision trees with different threshold values are also provided in later chapters and they are compared to trees with default threshold.

4.2 Benchmarking

Finding relation between application properties and garbage collection performance requires many programs with distinct characteristics. With that, it is possible to test programs with different garbage collector configurations and get best collector

for each. It means that the program should also stress memory extensively because otherwise no meaningful data can be obtained. Another requirement is that programs must reflect real-world programs.

4.2.1 Sorting algorithm benchmark suite

In this study, first a problem is defined which can have different implementations as solution. Integer sorting for this purpose is chosen because it is a common problem and has several algorithms as solution. Amongst common sorting algorithms, quicksort and merge sort are selected because they produce garbage in their implementations. Two common list implementations are also selected, ArrayList and LinkedList to experiment on. In each experiment, Benchmark application ran with changing following parameters: Sorting algorithm, list implementation and element count of sorted list.

4.2.2 DaCapo benchmark suite

Considering needs aforementioned, the DaCapo benchmark suite introduced by Blackburn et al is chosen for work [12]. To be precise, 9.12-bach version of suite is chosen as it is the most recent version and has more benchmarks than previous releases. Many of benchmarks provide different data size options. For each benchmark, the largest possible data size is chosen to produce most workload possible. On the other hand, some of benchmarks failed to run with OpenJDK 8 due to JDK changes after release of suite. As a result of this, those benchmarks are skipped. All benchmarks that are included in suite are listed in Table 4.1 with their usage status and data size used in this thesis. DaCapo suite provides two components which are used to facilitate this experiment. First, it is needed to report stable results for each benchmark. For example, it is possible to report average result after running a benchmark several times. Instead of that, proceeding with DaCapo suite's built-in warm-up mechanism is chosen. With a given maximum iteration parameter, DaCapo runs benchmark several times until results of executions converge or maximum number of iterations is reached. This allows to pick result of last execution only which is considered stable. Maximum iteration count is set as 20 and result convergence is enabled. Second component that is used is callback mechanism. DaCapo allows to inject a custom callback which runs

after each iteration. A custom callback is implemented to report execution time to a text file for processing later.

4.2.3 Benchmark driver

For collecting training data, it is needed to run benchmarks with all heap size and garbage collector combinations. For this purpose, a driver application, which runs the DaCapo suite as a separate process and records execution time and gc pause time results is implemented. Driver sets both JVM and DaCapo parameters. DaCapo parameters are discussed in previous section and JVM parameters will be discussed in next section.

4.2.4 JVM parameters

Java Virtual Machine can be run with parameters to behave differently. Following parameters are used by benchmark driver while starting benchmark processes.

-Xloggc:<logfilepath> When this options is enabled with file path, VM logs garbage collection events to file. Also it is possible to configure which events will be logged by setting other options.

-XX:+PrintGCApplicationStoppedTime This option enables VM to log total time of application pause due to garbage collection. It must be used with previous option. This option is used to report gc pause time.

-Xmx<size> Sets maximum heap size. It is possible to provide both a number and unit to define size. For example -Xmx1024m sets heap size to 1024 megabyte.

-Xms<size> Sets minimum heap size. Its usage is same with previous option.

-XX:+<collector> Instructs VM to use a certain garbage collector. Collector must be provided in argument. For example, -XX:+G1GC enables G1 GC.

4.2.5 Experimental setup

Microsoft Azure cloud platform virtual machines are used for experiments. Using a cloud platform allowed to create virtual machines as many as needed. Because of that it was possible to complete experiments in parallel. For DaCapo benchmark

Table 4.1 : DaCapo benchmark list with their status.

Benchmark	Usage	Used Data Size
avrora	Used	Large
batik	Failed	-
eclipse	Failed	-
fop	Used	Default
h2	Used	Huge
jython	Used	Large
luindex	Used	Default
lusearch	Used	Large
pmd	Used	Large
sunflow	Used	Large
tomcat	Failed	-
tradebeans	Used	Huge
tradesoap	Used	Huge
xalan	Used	Large

experiments, machines with 4 cores and 8 GB RAM which can be considered a relevant configuration for current real-world application loads are created. For sorting algorithm benchmark experiments, 1, 2, 4 and 8 cores machines are created. Ubuntu Server 16.04 is installed on machines as operating system. Finally OpenJDK8 is installed as Java runtime. For not interfering with results, no program is installed except mentioned and only benchmark driver and benchmarks ran while experimenting.

5. EXPERIMENTS AND EVALUATION OF GARBAGE COLLECTORS' PERFORMANCE

5.1 Sorting Algorithm Benchmark Evaluation

Sorting algorithm benchmark experiments are reported in this section. Firstly, effect of list size on measures is reported. Secondly, effect of algorithm and data structures is reported separately. Finally, effect of increasing core count of test machines is reported.

For reporting list size's effect on gc performance, results of sorting ArrayLists using merge sort with 4 core machine are chosen. First chart in Figure 5.1 shows GC pause time, execution time rate of small lists for all GCs. Lists with 1000000 elements are considered as small. Par and ParOld GC take least GC pause time and execution time. G1 GC also takes least gc pause time, however its execution time is slightly worse than Par and ParOld GC. Ser GC is slightly worse than G1 GC in both measures. CMS GC takes similar execution time to G1 GC but it clearly takes most GC pause time. Results of medium lists which have 3000000 elements showed in second chart. With increasing list size, Ser GC's gc pause time result becomes better relatively and Ser GC takes similar gc pause time to G1 GC while its execution time is slightly worse. Par and ParOld gc still have best result in both measures and CMS GC takes worst result. Third chart shows results of large lists which have 5000000 elements. With increasing list size, Par, ParOld and Ser GC's gc pause time measures became worse leaving G1 GC best collector in terms of gc pause time. Par and ParOld GC still have least execution time. Ser and CMS GC have most execution time and CMS GC also pauses application more than Ser GC.

Collection performance of sorting an ArrayList using merge sort and quicksort in 4 core test machine compared. Effect of algorithm on collection performance is shown in Figure 5.2. G1 GC's gc pause time performance is clearly different with merge sort and quicksort. Because of this difference, G1 GC takes most gc pause time in some cases and takes gc least pause time in a few cases while using quicksort. On the other hand, if merge sort is used, Ser GC takes least gc pause and CMS GC takes most gc

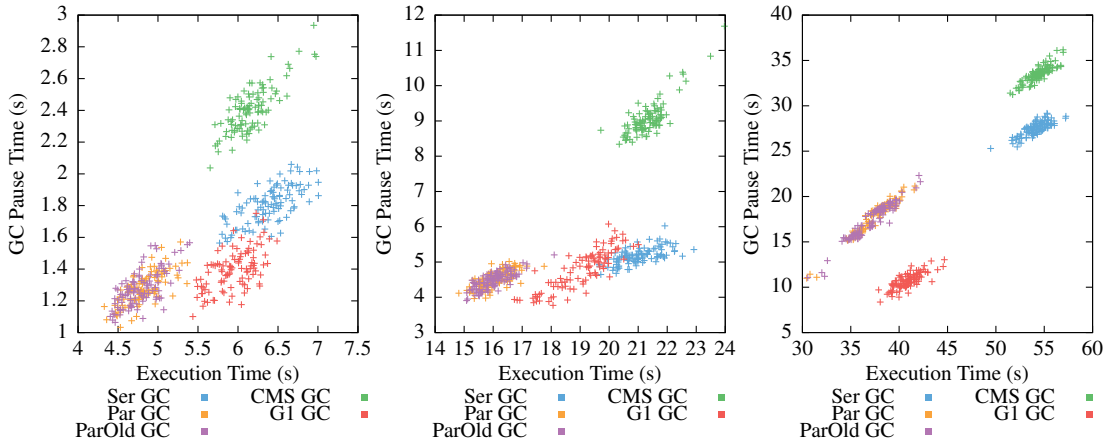

Figure 5.1 : GC performance by list size

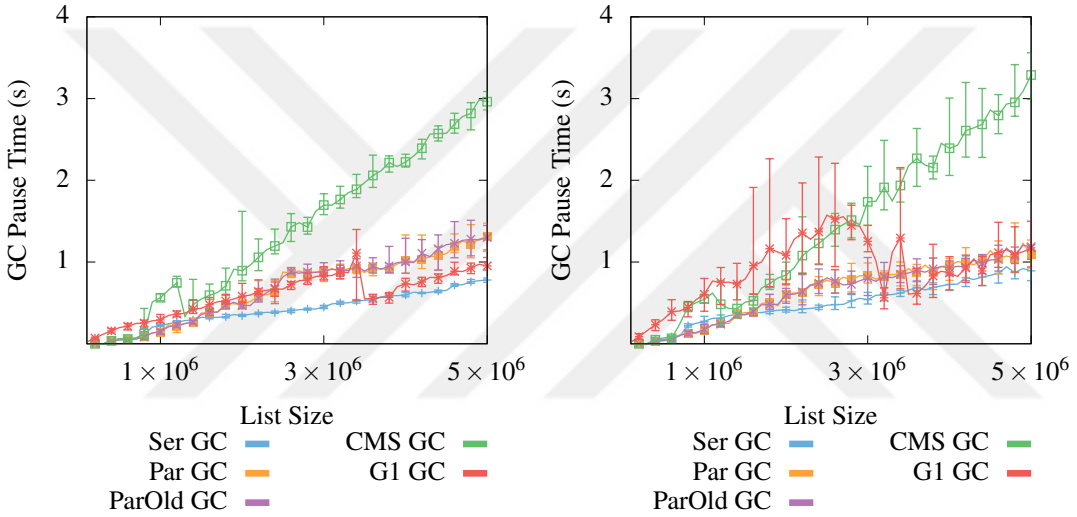

Figure 5.2 : Comparison of merge sort and quicksort respectively.

pause time in all cases. Execution time measures of merge sort and quick sort yielded similar results.

Merge sort results with 2 core machine to compare collection performance with different data structures are used. As seen in Figure 5.3, usage of different data causes gc performance change in both gc pause time and execution time measures. First chart shows that CMS GC takes most gc pause time in all cases if ArrayList and merge sort used. On the other hand, when LinkedList is used instead of ArrayList, Par, ParOld an Ser GC take more gc pause time than CMS GC starting from list size of 3500000 as illustrated in second chart. Considering execution time measure, results also change when different data structures used. When ArrayList used, Ser GC receives best result in all cases. Par and ParOld GC also receive best result for small list sizes. However

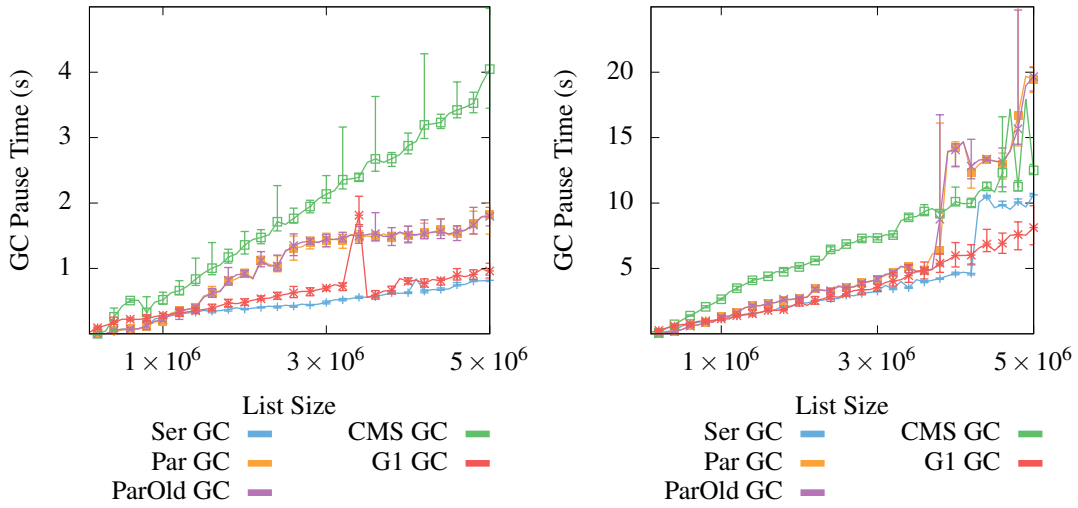

Figure 5.3 : Comparison of ArrayList and LinkedList respectively

when LinkedList used instead of ArrayList, Par and ParOld GC take least execution time for small and medium list sizes while CMS GC takes least execution time for large list sizes.

Effect of core count on GC performance is investigated. Findings are reported in Figure 5.4 using ArrayList and quicksort results, however similar results are also observed in other environments. First chart in Figure 5.4 shows that Ser GC does not receive better gc pause time as core count increases. G1 GC's gc pause time decreases as core count increases, however it receives worse gc pause time when core count is 8. CMS GC's gc pause time performance is better with 4 and 8 core machines and is worst with 2 core machine. Par and ParOld GC's performance increase with core count. Results for Ser and CMS GC are also similar in execution time measure as shown in second chart. G1 GC's performance increases greatly with core count giving best result with 8 core machine. Par and ParOld GC performance increase slightly with core count.

Sorting algorithm benchmarks are not suitable for property extraction because they are simple applications. On the other hand, experiment results show that garbage collection performance are related to program implementation and different performance results can be obtained with different implementations which do same operations. Therefore this work is continued by DaCapo benchmark experimenting to elaborate on this behaviour.

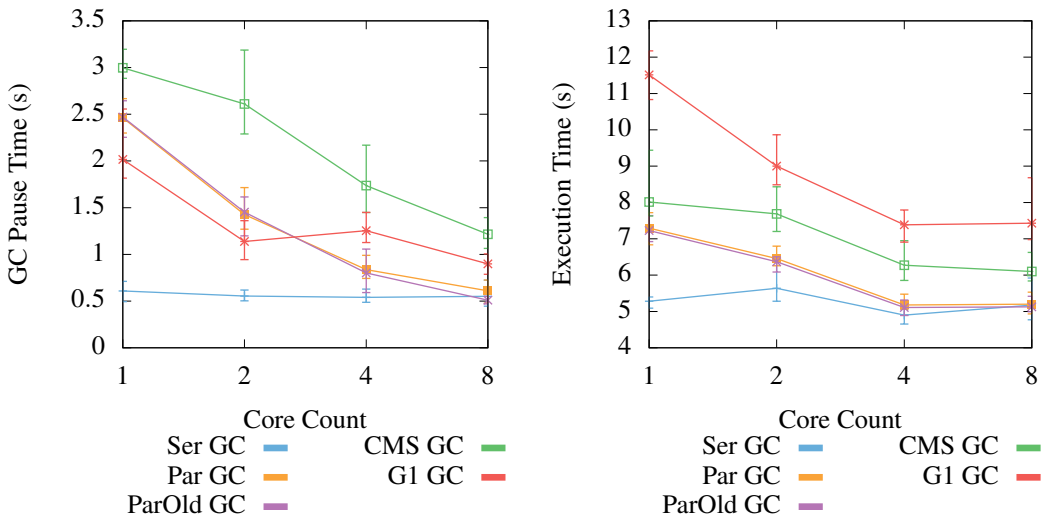

Figure 5.4 : GC performance of quicksort by core count with ArrayList.

5.2 DaCapo Benchmark Evaluation

DaCapo benchmarks ran using environment setup aforementioned. Each benchmark ran with 5 different garbage collectors and 4 different heap sizes: 1024, 2048, 3072, 4096 megabytes respectively. Benchmark driver reported results for examination.

As mentioned before, some benchmarks failed in all cases. In addition, it is observed that some benchmarks failed with 1024MB heap size due to insufficient memory. These are, avrora benchmark with Ser GC and h2 benchmark with CMS GC, Par GC and ParOld GC. Summary of best garbage collector configurations are reported in Table 5.1. It is clearly seen that Parallel and Parallel Old collectors dominate most cases. It should also be noted that Par and ParOld collectors are very similar implementations. Other collectors are best performing in remaining cases. In Table 5.2, summary of acceptable performances for each configuration reported. Parallel and Parallel Old collectors had acceptable performance most of time. On the other hand, CMS GC always performed bad compared to others in terms of throughput or responsiveness, so it remained unacceptable in all cases. When performance threshold changes, percentage of acceptable benchmark percentage also changes as shown in Figure 5.5. For training data of decision tree learners, it is desired to have a decent amount of acceptable cases. Otherwise, algorithms won't be able to find patterns for acceptable cases. So threshold must be chosen as high as possible. On the other hand, high threshold means performance gap between best performing garbage collector

and other acceptable garbage collectors will be larger. Therefore programs evaluated by predictor won't be able to benefit from potential performance increase. There is a trade-off between having more success rate with decision trees and having better performance increase for successful predictions. Considering these, performance threshold is chosen as 120% which marks 35.64% of cases as acceptable. However, in later chapters, decision trees with different performance thresholds are also introduced and their results are reported.

Considering CMS GC and G1 GC are introduced later than the rest to address gc pause time problems, It is unexpected to not find them most responsive in most cases. On the other hand, it is observed that the best collector changes with different programs so previous assumption for relation between program properties and gc performance is correct. Therefore this work is continued by analyzing programs to find properties which might have been influenced garbage collector performance.

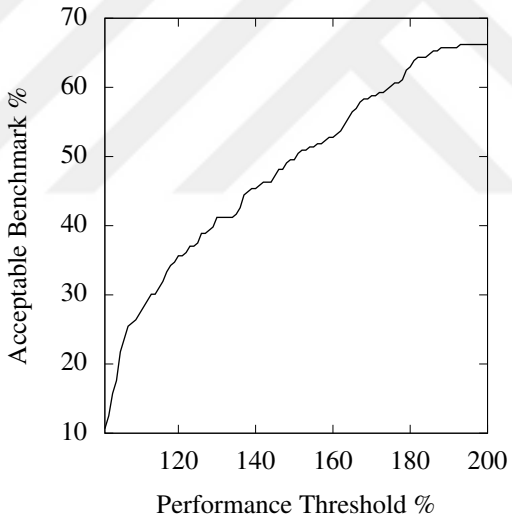

Figure 5.5 : Change of acceptable cases with performance threshold.

Table 5.1 : Best garbage collectors for each benchmark.

Benchmark	1024 MB Heap		2048 MB Heap		3074 MB Heap		4096 MB Heap	
	Responsive	Throghput	Responsive	Throghput	Responsive	Throghput	Responsive	Throghput
avrora	Parallel	ParOld	Parallel	Parallel	ParOld	CMS	Parallel	CMS
fop	Parallel	Parallel	ParOld	ParOld	ParOld	Serial	ParOld	Serial
h2	G1	G1	Parallel	ParOld	ParOld	Parallel	Parallel	ParOld
jython	ParOld	ParOld	Parallel	Serial	ParOld	Parallel	Parallel	ParOld
luindex	Serial	Parallel	Serial	CMS	Serial	ParOld	Serial	Parallel
lusearch	G1	Parallel	Serial	Parallel	Parallel	Serial	ParOld	ParOld
pmd	Parallel	Serial	ParOld	Parallel	Parallel	ParOld	ParOld	CMS
sunflow	ParOld	Serial	Parallel	G1	ParOld	G1	Parallel	G1
tradebeans	Parallel	Parallel	Parallel	ParOld	G1	CMS	Parallel	Parallel
tradesoap	ParOld	ParOld	Parallel	Parallel	Parallel	Parallel	Parallel	Parallel
xalan	ParOld	Serial	ParOld	ParOld	Parallel	Parallel	ParOld	ParOld

Table 5.2 : Count of benchmarks with acceptable performance for each garbage collector and heap size.

GC	1024MB	2048MB	3072MB	4096MB
CMS GC	0	0	0	0
G1 GC	3	0	2	1
Par GC	6	10	9	8
ParOld GC	5	8	9	8
Ser GC	2	3	1	2

6. OBJECT ORIENTED METRICS TO BE USED IN GARBAGE COLLECTION SELECTION

In this chapter, a collection of program properties are introduced that are probably influential on which garbage collector performs best. Analyze process to obtain these properties are also shared. These properties are split into 2 categories which will be discussed in sections below.

6.1 Metrics Based on Static Properties

Static properties can be described as properties which can be obtained by analyzing source code or byte code for Java. Blackburn et al share Chidamber and Kemerer metrics of benchmarks in their paper [3, 12]. However, these results can't be used in this research because they are extracted for a previous version of DaCapo. Therefore it is decided to analyze source code of DaCapo 9.12-bach version. While DaCapo is open source, its source distribution doesn't contain source code of all files. It uses many libraries which are included as jar file and these libraries contain most of benchmarks' code. These libraries' source code are manually downloaded if exact version used in benchmark is available. These sources are also processed. After collection, source code is analyzed using JHawk[1] tool. Aside from Chidamber and Kemerer metrics, JHawk can measure many class level metrics. Several metrics are chosen as properties which are discussed below. More detailed information about these metrics and their measurement methods can be found in JHawk's guide[2].

- AVCC - Average Cyclomatic Complexity: AVCC is average cyclomatic complexity of all methods of class. Cyclomatic complexity is calculated using method introduced by McCabe [13].

[1] http://www.virtualmachinery.com/jhawkprod.htm
[2] http://www.virtualmachinery.com/jhawkmetrics.htm

- TCC - Total Cyclomatic Complexity: Sum of McCabe cyclomatic complexity of all methods in class.
- MAXCC - Maximum Cyclomatic Complexity: Cyclomatic complexity of method which has most cyclomatic complexity in class.
- UWCS - Unweighted Class Size: Total number of methods and class variables
- INST - Instance Variables: Measured by counting total number of instance variables in class.
- PACK - Packages imported: Measured by counting total number of packages imported in class.
- MOD - Modifiers: Measured by counting accessibility keywords like private or public used in class.
- NOS - Number of Statements: Total number of statements found in class. JHawk counts declarations, control statements and assignments to measure this metric as an alternative to lines of code.
- NCO - Number of Commands: Measured by counting methods that don't return a value in class.
- NQU - Number of Queries: Measured by counting methods that return a value in class.
- EXT - External Methods: Measured by counting external method calls by class.
- LMC - Local Methods Called: Measured by counting local method calls by class.
- RFC - Response For a Class: RFC is total number of methods which can be called from the class. It is one of Chidamber and Kemerer metrics.
- CBO - Coupling between object classes: When an instance of a class accesses an instance of another class, these classes are considered as coupled. Therefore CBO for a class is defined as amount of other distinct classes which it is coupled. It is one of Chidamber and Kemerer metrics.

- FOUT - Fan Out: FOUT is similar to CBO, but it only counts classes which measured class reference to.
- FIN - Fan In: FIN is similar to CBO, but it only counts classes which reference measured class.
- MPC - Message Passing Coupling: Measured by counting messages passed between different classes.
- DIT - Depth of Interitance Tree: DIT is total number of ancestor classes which class is inherited. It is one of Chidamber and Kemerer metrics.
- RR - Reuse Ratio: RR is calculated by dividing count of superclasses by total number of classes in hiearchy.
- SR - Specialization Ratio: SR is calculated by dividing count of classes which inherited from measured class by count of classes which measured class inherit.
- LCOM - Lack of Cohesion in Methods: LCOM is calculated considering shared instance variable count in each method pair in the class. It is one of Chidamber and Kemerer metrics.
- COH - Cohesion: COH is calculated considering shared instance variables between methods.

Described metrics are measured class-level. To obtain a metric per benchmark, metric values of all classes that are actually loaded in benchmark execution are summed as also done by Blackburn et al [12]. Some programs may contain many test code or code of features unused in execution. Therefore this approach allows to have more precise definition of program executed. Standard libraries' classes which are included in JDK are also skipped as they are used in all benchmarks and don't represent program characteristics.

Aforementioned static properties are measured for all benchmarks. Properties that measure class size are reported in Table 6.1. Properties that measure coupling of classes are reported in Table 6.2. Finally properties that measure complexity of classes are reported in Table 6.3.

Table 6.1 : Class size properties of benchmarks.

Benchmark	UWCS	INST	PACK	MOD	NOS	NCO	NQU	EXT	LMC	RFC
jython	23221	3509	6966	1102	140902	4473	15239	450	2671	20162
tradebeans	10368	3296	5526	484	57715	2912	4160	145	1551	7217
lusearch	8640	2696	7619	1112	52606	3279	2665	267	1197	6211
xalan	23638	13254	9237	1304	83236	4100	6284	201	1943	10585
luindex	8640	2696	7619	1112	52606	3279	2665	267	1197	6211
avrora	14183	5325	6262	2641	53372	5226	3632	571	1795	9429
fop	43735	16141	61608	4937	199803	12774	14820	1144	5011	28738
h2	24084	6677	17205	1692	185377	9642	7765	347	4140	17754
pmd	9934	2665	8757	1133	52692	3024	4245	275	1843	7544
sunflow	2146	850	3797	290	14684	640	656	12	192	1308
tradesoap	10368	3296	5526	484	57715	2912	4160	145	1551	7217

Table 6.2 : Class coupling properties of benchmarks.

Benchmark	CBO	FOUT	FIN	MPC	DIT	RR	SR	LCOM	COH
jython	12127	6591	5985	450	2254	328.94	758.55	286.37	141.07
tradebeans	7665	4062	4059	145	890	136.17	288.74	132.10	101.35
lusearch	8530	4412	4250	267	1729	297.77	424.32	227.73	230.92
xalan	10079	5250	5215	201	2581	411.63	662.15	388.08	269.88
luindex	8530	4412	4250	267	1729	297.77	424.32	227.73	230.92
avrora	11147	5835	5487	571	3105	591.25	606.83	453.67	495.57
fop	28955	15266	14289	1144	8397	1222.70	2067.96	1150.32	807.93
h2	12428	6655	6303	347	3568	533.96	749.87	347.54	258.64
pmd	8714	4675	4651	275	2180	349.92	647.96	226.85	184.74
sunflow	1767	956	861	12	268	18	16	51.53	66.91
tradesoap	7665	4062	4059	145	890	136.17	288.74	132.10	101.35

Table 6.3 : Class complexity properties of benchmarks.

Benchmark	AVCC	TCC	MAXCC
jython	1618.64	49312	4102
tradebeans	978.91	16324	3202
lusearch	1767.98	12663	3675
xalan	2523.61	23294	5916
luindex	1767.98	12663	3675
avrora	2418.18	14096	3761
fop	7492.62	55164	16011
h2	3327.70	39371	8603
pmd	1972.31	16353	4131
sunflow	539.47	3315	1295
tradesoap	978.91	16324	3202

6.2 Metrics Based on Dynamic Properties

Static properties can't define a program totally. For example, allocation statistics can't be determined as it is mostly decided in runtime. Therefore in addition to aforementioned, following properties are proposed to define a program.

- Object Allocation: Allocation of objects which are not from standard libraries is considered. Because these objects are possibly more complex than others, this property is chosen.
- Array Allocation: Arrays are common in programs and they can also allocate large parts in heap, so this property is chosen.
- Char Array Allocation: In experiments, it is found that char array allocation accounts for most of total allocations. So this property is included.
- Linked List Node Allocation: As mentioned before, Eran and Petrank showed that linked lists are not efficient data structures for multithread garbage collectors [7]. Therefore, this property included to assess this claim.

All properties mentioned are measured as percentage of total number of allocations. Absolute count is not included because it could be misleading as these measures will be naturally high in programs which have long execution time.

Aforementioned properties are measured with a single run of benchmarks. Allocation statistics are not related to garbage collector used so this execution is done with default

JVM parameters. To acquire dynamic properties, YourKit Java Profiler[3] with version 2017.02-b66 is used. Profiler allows to count all allocated objects precisely. A slight change is added to DaCapo benchmark suite to profile it correctly. 30 seconds of thread sleep injected in the main method, so profiler can be attached to process.

Dynamic properties of benchmarks are presented in Table 6.4. It is possible to see some characteristics of benchmarks. Linked list is not common in all benchmarks except avrora. On the other hand, avrora benchmark's object allocation percentage is very low, which means it heavily uses Java standard libraries. Sunflow benchmark is another marginal example. It heavily allocates non-standard objects while not using arrays, linked lists and char arrays much. Most char array and array usage is in xalan benchmark.

Table 6.4 : Dynamic properties of benchmarks.

Benchmark	Object %	Array %	Char %	Linked List %
avrora	4.87	22.32	22.16	22.16
fop	43.34	29.41	20.13	20.13
h2	40.25	34.34	20.13	20.13
jython	62.81	28.32	9.34	9.34
luindex	36.21	28.49	21.60	21.60
lusearch	47.51	33.71	18.61	18.61
pmd	36.13	28.73	13.50	13.50
sunflow	97.05	2.90	0.01	0.01
tradebeans	35.59	36.20	24.17	24.17
tradesoap	32.95	31.51	22.85	22.85
xalan	49.24	37.79	34.37	34.37

[3] https://www.yourkit.com/java/profiler/features/

7. USING DECISION TREES ON GARBAGE COLLECTOR SELECTION

In previous chapters, implemented sorting algorithm benchmark and the DaCapo benchmark suite is experimented with and best garbage collectors for benchmarks are determined. For sorting benchmark, 2 data type, 2 algorithm, heap size ranging from 1GB to 3GB and list size ranging from 250000 to 5000000 are used. For DaCapo, 4 different heap sizes are used and 5 garbage collection configuration are also used as attributes. Also other attributes which include 22 static and 4 dynamic metrics to define program properties are introduced. This data constructed the training data for decision tree learner. Weka machine learning software is used as it is popular in academic researches and also offers several algorithms which are easy to use and test [14].

7.1 Sorting Benchmark Garbage Collection Prediction

As mentioned in previous chapters, sorting benchmark is not suitable for program property extraction and it can't be used to obtain generalized garbage collection performance predictor for object oriented programs. However, decision trees are created to predict sorting algorithms' garbage collection performance using heap size, data type and data size information.

Performance rate of sorting benchmarks are calculated by dividing gc pause time by execution time. This value expected to be lower as it means program is more responsive. garbage collectors are marked as acceptable and unacceptable using 120% performance threshold.

Generated decision trees are reported in Table 7.1. Random forest algorithm generated most successful tree with a success rate of 91.76%. Also J48, Random tree and REPTree algorithms created decision trees with good success rate. On the other hand, these trees are quite large which are ranging from 482 to 2408.

Table 7.1 : Comparison of decision tree learner algorithms for sorting benchmark.

Algorithm	Tree Size	Used Property Count	Success Rate %
Decision stump	3	1	61.08
Hoeffding tree	10	2	81
J48	566	5	90.43
Random forest	100 Trees	5	91.76
Random tree	2408	5	89.09
REPTree	482	5	88.49

7.2 DaCapo Benchmark Garbage Collection Prediction

7.2.1 Best garbage collector predictor

First of all, using provided decision tree learner algorithms, decision trees which predict best garbage collectors are created. All of available metrics are used in this experiment. Summary of decision trees is reported in Table 7.2. Most successful tree is generated by decision stump algorithm with 40.91% success rate. It only offers Parallel or Parallel Old garbage collector depending on Object allocation percentage metric. Hoeffding tree does not consider any metric useful and only offers Parallel garbage collector for all cases.

Table 7.2 : Comparison of decision tree learner algorithms.

Algorithm	Tree Size	Used Property Count	Success Rate %
Decision stump	4	1	40.9091
Hoeffding tree	1	0	39.7727
J48	33	5	29.5455
Random forest	100 Trees	27	27.2727
Random tree	133	11	34.0909
REPTree	3	1	35.2273

Decision trees are also created to predict best garbage collector, using only static properties. This is useful because it allows to predict garbage together without running program altogether. Heap size and dynamic properties are removed from training data and algorithms are evaluated. Most successful tree is REPTree with 42.04% success rate. REPTree first checks PACK metric. If imported packages count is less than 6614, it checks if LCOM is less than 91.81. If so, Parallel GC must be used if pause time is more important and G1 GC must be used if execution time is more important. If LCOM is greater than or equal to 91.81, Parallel GC must be used. If PACK is greater

than or equal to 6614, It checks if AVCC is greater than or equal to 2925.65 and NOS is greater than or equal to 52649. If so, Parallel Old GC must be used. Otherwise, Serial GC must be used if pause time is important or Parallel GC must be used if execution time is important. Summary of decision trees which uses only static properties to predict best garbage collector is reported in Table 7.3

Overall, success rate of decision trees which predict best garbage collector was below expectations. This result is mostly due to similar garbage collection results. In several cases, the best garbage collector outperforms the second by only a few milliseconds so this result is not distinctive for decision trees. However, it is expected to have better success with decision trees which predict garbage collector performance acceptance.

Table 7.3 : Comparison of decision tree learner algorithms with only static properties.

Algorithm	Tree Size	Used Property Count	Success Rate %
Decision stump	3	1	36.3636
Hoeffding tree	1	0	39.7727
J48	13	4	37.5
Random forest	100 Trees	22	32.9545
Random tree	35	7	37.5
REPTree	13	5	42.0455

7.2.2 Garbage collector performance predictor

Work is continued by creating a decision tree which predict garbage collector performance acceptance. J48 decision tree, which is an implementation of C4.5 decision tree introduced by Quinlan is chosen [15]. J48 can generate pruned or unpruned trees. Pruned tree is generated, because many metrics are given to decision tree learner and it is expected for some of them to be irrelevant. Therefore, metrics which don't affect garbage collection performance will be pruned and not included in decision tree. Other options are left as default.

Decision tree generated as aforementioned. First of all, Parallel GC usage is considered acceptable for all cases and no relation is found between Parallel GC performance and chosen metrics. On the other hand, CMS GC is not accepted for any case naturally.

Serial GC performance is related to NOS and Object Percentage metrics as shown in Figure 7.1. If NOS is less than or equal to 52606 and Object Percentage is less than or equal to 40.25%, it is acceptable to use Serial GC.

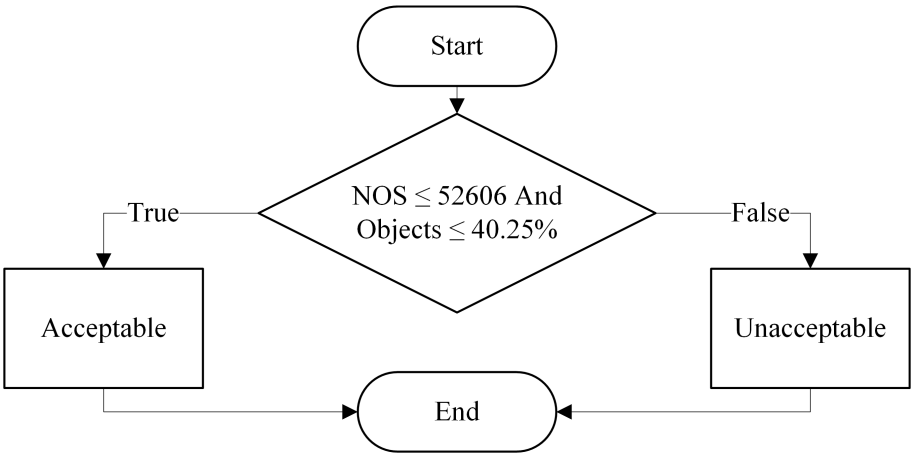

Figure 7.1 : Decision tree generated for Serial GC evaluation.

G1 GC performance prediction is determined as shown in Figure 7.2. If Array Percentage is less than or equal to 31.51%, G1 GC usage is unacceptable. Otherwise, if Heap size is less than or equal to 1024MB, G1 GC can be used. If Array Percentage is more than 31.51% and Heap Size is more than 1024MB, LCOM should be less than or equal to 132.1 for G1 GC usage is to be acceptable.

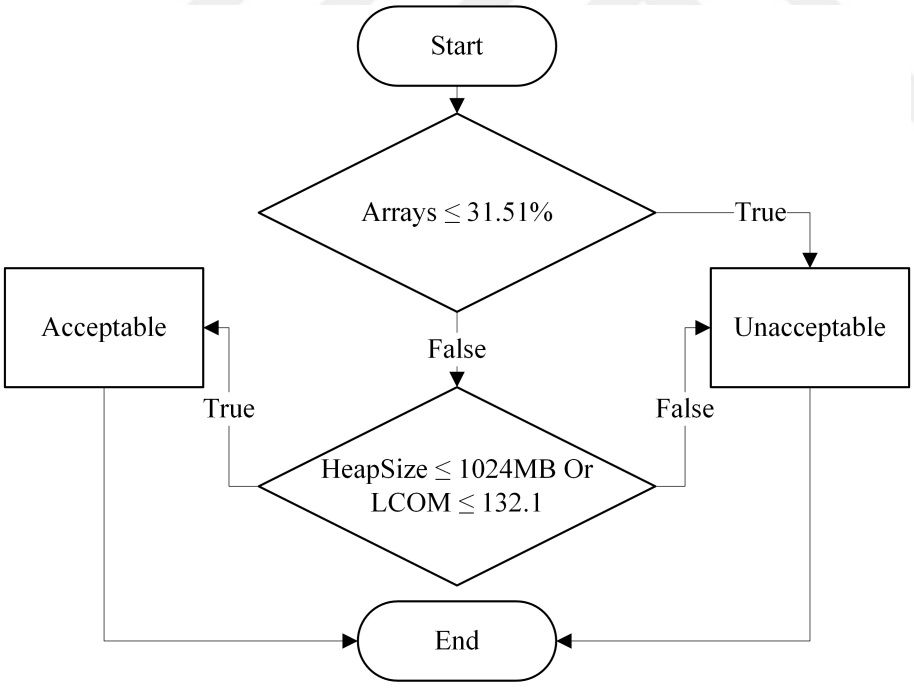

Figure 7.2 : Decision tree generated for G1 GC evaluation.

Parallel Old GC performance is predicted using 4 metrics as shown in Figure 7.3. If Heap Size is more than 1024MB or Object Percentage is more than 47.51%, Parallel Old GC is acceptable. Otherwise, AVCC should be less than or equal to 1767.98 and Array Percentage should be less than or equal to 31.51%.

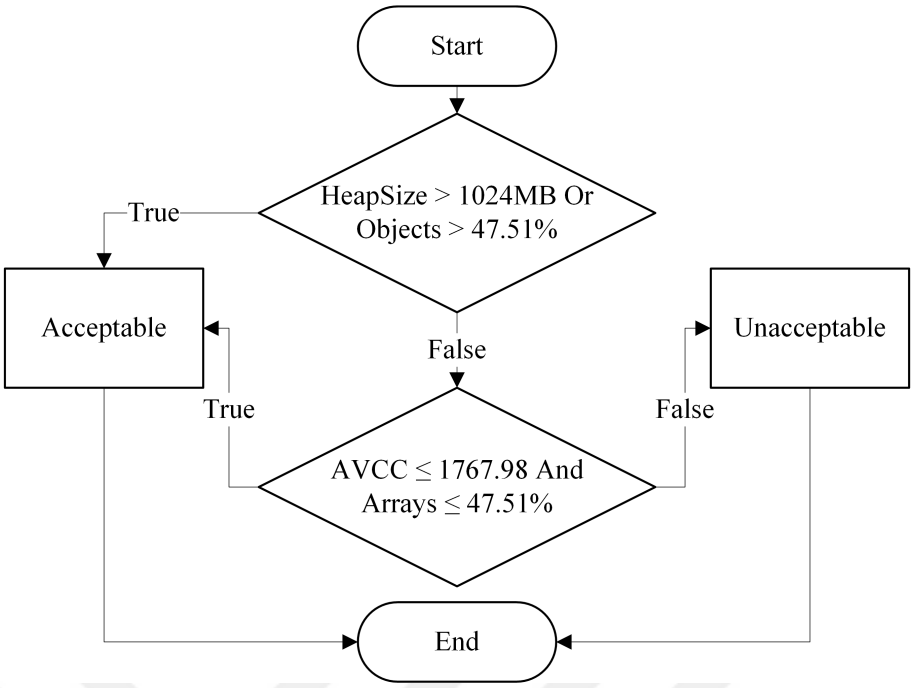

Figure 7.3 : Decision tree generated for Parallel Old GC evaluation.

Decision tree is validated using 10-folds cross validation. Decision tree classifies 166 of 216 instances correctly with a success rate of 76.85%. Confusion matrix is also shown in Table 7.4. Incorrectly classifying instances as acceptable is particularly unwanted. 30 instances classified as acceptable while they are actually unacceptable. On the other hand, 20 instances are classified as unacceptable while they are actually acceptable. However these incorrect classifications are less important than aforementioned. For classifications as acceptable, precision is 0.655 while recall is 0.74. For classifications as unacceptable, precision is 0.845 while recall is 0.784. Recall rates for both classifications are similar, so predictions' completeness are similar. On the other hand, precision for classification as unacceptable is much higher than classification as acceptable. Therefore, predictor is more accurate while detecting unacceptable cases.

Table 7.4 : Confusion matrix of J48 decision tree.

	Classified Acceptable	Classified Unacceptable
Actually Acceptable	57	20
Actually Unacceptable	30	109

Experiment is repeated with changing algorithm to Random forest. Random forest algorithm generates a forest from complex trees that uses all metrics provided. Success rate of this decision tree is 79.17%, which is slightly better than J48 tree. However it

requires more metrics than J48 algorithm for prediction. Random forest algorithm's tree count can be increased to obtain more successful results. Experiment repeated with different forest sizes, ranging from 100 which is default to 500. However larger forests didn't provide an improvement. Success rate of Random forest with 500 trees remained at 77.31%.

120% is chosen as performance threshold for decision tree. To observe threshold effect on decision trees, threshold is changed for different decision trees. Success rates of decision trees with different thresholds are reported in Table 7.5. As threshold increases, decision trees start to predict CMS GC performance depending on metrics. It is observed that, decision trees which have better success rates can be obtained when threshold is chosen as 130% or 140%. It also means that garbage collectors which are unacceptable with lower thresholds will be acceptable. Therefore, threshold can be chosen considering performance requirements of environment.

Table 7.5 : Comparison of success rates with different performance thresholds.

Algorithm	110%	120%	130%	140%	150%
Decision stump	70.37	64.81	75	71.76	63.43
Hoeffding tree	72.69	65.28	61.11	61.57	62.04
J48	76.85	76.85	81.48	83.80	80.10
Random forest	74.54	79.17	82.87	84.72	83.80
Random tree	74.54	72.22	74.51	81.02	73.15
REPTree	78.24	80.09	85.65	85.65	84.26

As implied before, excluding dynamic properties is also beneficial for discarding single run. Because of this, performance prediction experiment is repeated with using only static properties. J48 algorithm generated a simple tree which only uses properties to predict Serial GC performance. If NOS is less than or equal to 52606 and LCOM is more than 132.1, Serial GC usage is acceptable. J48 considers Parallel GC and Parallel Old GC as always acceptable. On the other hand, G1 GC and CMS GC is always unacceptable. This experiment increased J48 algorithm's success rate to 82.41%. Using static properties, performance predictor is also generated by other algorithms. Random tree algorithm achieves 79.17% success rate. It uses AVCC, EXT, FOUT, LCOM, MAXCC, NOS, NQU and UWCS metrics. REP tree also achieves 81.94% success rate but it does not consider any property useful and only accepts Parallel and Parallel Old GCs.

Another approach to increase success rate is feature selection. Weka allows correlation based feature selection to detect most useful properties for garbage collection performance prediction. Useless properties can be removed and decision trees can be generated with new dataset. Correlation based feature selector evaluates following properties as most useful: MOD, EXT, MPC, COH, RR. On the other hand, following properties are evaluated as useless: NQU, INST, TCC, NOS, UWCS, RFC, Heap size, MAXCC, LMC, CBO. Decision trees are regenerated after deletion of useless features. Feature selection resulted in performance increase with Hoeffding tree, J48, Random tree and REPTree algorithms as shown in Figure 7.4. For J48 algorithm, success rate is increased to 82.41%. On the other hand, decision tree used MPC metric for only Serial GC evaluation. Parallel and Parallel Old GCs are evaluated as acceptable while CMS and G1 GCs are evaluated as unacceptable for all cases.

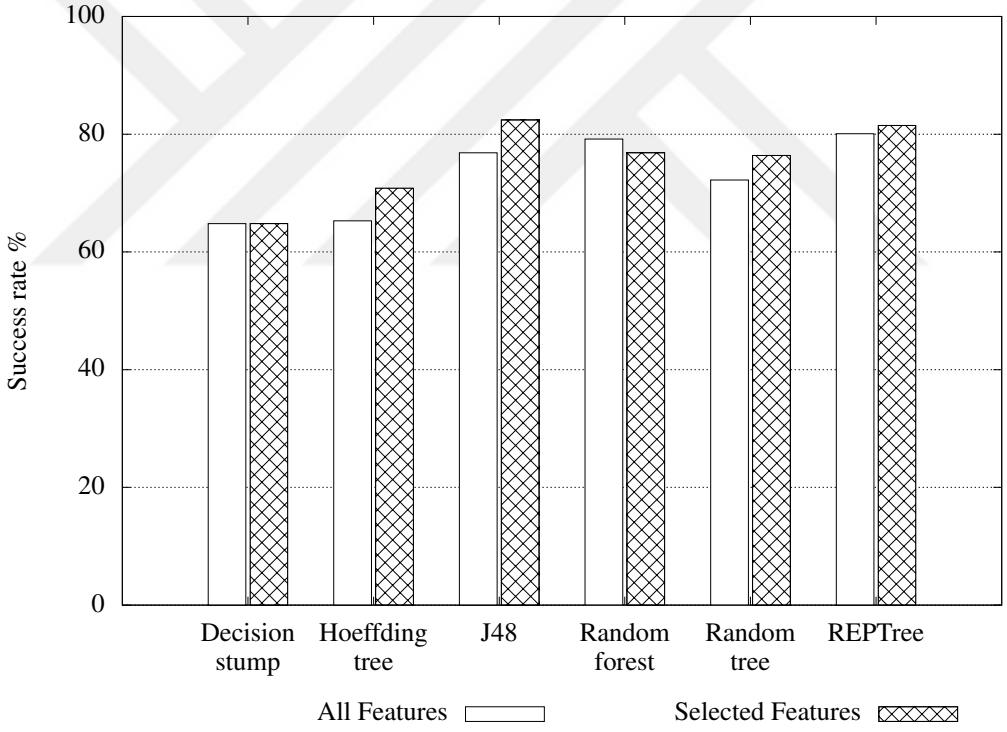

Figure 7.4 : Change of success rate after feature selection.

8. CONCLUSIONS AND FUTURE WORK

In this thesis, a relation between program properties and garbage collector is proposed. To prove this relation, a program set is experimented with to report its garbage collection performance. Afterwards, programs are analyzed and a decision tree is built with data acquired. In this chapter, Our findings are summarized and conclusions are reported.

Huge amount of data is needed for decision tree learner to yield efficient results. This data can be obtained by only experimenting with many programs. Choosing programs one by one is not feasible as running them alone requires too much manual work. So it is reasonable to use benchmark suites like the DaCapo benchmark suite. DaCapo facilitated research a lot by providing programs which are easy to run. It is also used in many academic researches so it was possible to learn from their findings. On the other hand, last version of DaCapo was not released recently and because of that, some benchmarks failed to run with recent JVMs. So smaller data size is obtained than possible. However, there is not any more recent benchmark suite to replace DaCapo. Another issue is, DaCapo is a benchmark suite and is not developed primarily for code analysis, so it required some manual work to obtain program properties. Chidamber and Kemerer metrics of suite are given in its paper but these metrics are obsolete as another version is released. These metrics are measured for latest version and reported in this thesis, so researchers can make use of results in future. There are some works that provide program collection for static analysis, however they can't be executed or they can't stress system memory as required. Considering these, the DaCapo benchmark suite remains as most suitable choice for garbage collection research. A benchmark suite that addresses all issues mentioned and covers huge amount of programs would greatly help future work on this area. However this kind of work is non-trivial and beyond the scope of thesis.

After extensively testing DaCapo benchmarks with different garbage collectors, best garbage collectors in two criteria for each benchmark are reported. Par GC and

ParOld GC performed good in most cases. A naive approach would be to assume garbage collectors that are introduced later will outperform others. However it proved otherwise. G1 GC performed good in several cases while CMS GC didn't performed good in any cases. Similarities and differences are found with related work before. Therefore a good approach would be to test program with all garbage collectors regardless of collectors' release date and promoted features.

Many static properties of programs including Chidamber and Kemerer metrics are used. It is found out that some of metrics are related to memory usage of program while others are not. It is expected because in fact these metrics are introduced to define facilitate development and maintenance process. On the other hand, some metrics were useful and they are used in decision trees.

J48 decision tree learner is used to construct garbage collector performance predictor. Success rate of decision tree is 76.85%. It is proposed that success rate can be improved by experimenting with a larger training dataset which contains more programs and defining properties. This research area is still open to many possible researches in future.

REFERENCES

[1] **Jones, R.**, **Hosking, A. and Moss, E.** (2012). *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Applied Algorithms and Data Structures, Chapman & Hall.

[2] **Singer, J.**, **Brown, G.**, **Watson, I. and Cavazos, J.** (2007). Intelligent Selection of Application-specific Garbage Collectors, *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, ACM, New York, NY, USA, pp.91–102, http://doi.acm.org/10.1145/1296907.1296920.

[3] **Chidamber, S.R. and Kemerer, C.F.** (1994). A Metrics Suite for Object Oriented Design, *IEEE Trans. Softw. Eng.*, *20*(6), 476–493, http://dx.doi.org/10.1109/32.295895.

[4] **Lengauer, P. and Mössenböck, H.** (2014). The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors, *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, ACM, New York, NY, USA, pp.111–122, http://doi.acm.org/10.1145/2568088.2568091.

[5] **Libič, P.**, **Bulej, L.**, **Horký, V. and Tůma, P.**, (2015). Estimating the Impact of Code Additions on Garbage Collection Overhead, Springer International Publishing, Cham, pp.130–145, https://doi.org/10.1007/978-3-319-23267-6_9.

[6] **Lengauer, P.**, **Bitto, V.**, **Mössenböck, H. and Weninger, M.** (2017). A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, ACM, New York, NY, USA, pp.3–14, http://doi.acm.org/10.1145/3030207.3030211.

[7] **Eran, H. and Petrank, E.** (2013). A Study of Data Structures with a Deep Heap Shape, *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, ACM, New York, NY, USA, pp.2:1–2:8, http://doi.acm.org/10.1145/2492408.2492413.

[8] **Carpen-Amarie, M.**, **Marlier, P.**, **Felber, P. and Thomas, G.** (2015). A Performance Study of Java Garbage Collectors on Multicore Architectures, *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, ACM,

New York, NY, USA, pp.20–29, http://doi.acm.org/10.1145/2712386.2712404.

[9] **Gidra, L.**, **Thomas, G.**, **Sopena, J. and Shapiro, M.** (2011). Assessing the Scalability of Garbage Collectors on Many Cores, *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, ACM, New York, NY, USA, pp.7:1–7:5, http://doi.acm.org/10.1145/2039239.2039249.

[10] **Detlefs, D.**, **Flood, C.**, **Heller, S. and Printezis, T.** (2004). Garbage-first Garbage Collection, *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, ACM, New York, NY, USA, pp.37–48, http://doi.acm.org/10.1145/1029873.1029879.

[11] **Lindholm, T.**, **Yellin, F.**, **Bracha, G. and Buckley, A.** (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*, Addison-Wesley Professional, 1st edition.

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

[13] **McCabe, T.J.** (1976). A Complexity Measure, *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320.

[14] **Witten, I.H.**, **Frank, E.**, **Hall, M.A. and Pal, C.J.** (2016). *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition.

[15] **Quinlan, R.** (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA.

APPENDICES

APPENDIX A : Benchmark driver source code
APPENDIX B : Execution results of benchmarks
APPENDIX C : Object allocation distribution of benchmarks

APPENDIX A

Main.java

```
package com.szkg.gcbenchmarkrunner;

public class Main {

  public static void main(String[] args) {
    new DaCapoRunner().Run(args);
  }
}
```

DaCapoRunner.java

```
package com.szkg.gcbenchmarkrunner;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.*;
import java.util.Map.Entry;

public class DaCapoRunner {

  private String[] collectors = new String[]{
      "UseSerialGC",
      "UseParallelGC",
      "UseG1GC",
      "UseConcMarkSweepGC",
      "UseParallelOldGC"
  };

  private Map<String, String> benchmarks =
      new HashMap<>();

  private int[] heapSizes = new int[]{
      1024 * 1,
      1024 * 2,
      1024 * 3,
      1024 * 4
  };
```

```
private static final String resultFile = "Result.txt";

public void Run(String[] args) {

  try {

    benchmarks.put("avrora", "large");
    benchmarks.put("batik", "large");
    benchmarks.put("fop", "default");
    benchmarks.put("h2", "huge");
    benchmarks.put("jython", "large");
    benchmarks.put("luindex", "default");
    benchmarks.put("lusearch", "large");
    benchmarks.put("pmd", "large");
    benchmarks.put("sunflow", "large");
    benchmarks.put("tradebeans", "huge");
    benchmarks.put("tradesoap", "huge");
    benchmarks.put("xalan", "large");

    Files.createDirectories(Paths.get("Logs"));

    String jarName = "dacapo-9.12-bach.jar";

    writeResult(resultFile,
        String.format("%s\t%s\t%s\t%s\t%s",
            "HeapSize",
            "Benchmark",
            "Collector",
            "PauseTime",
            "ExecutionTime"));

    for (int heapSize : heapSizes) {
      for (Entry<String, String> benchmark :
          benchmarks.entrySet()) {
        for (String collector : collectors) {
          runIteration(jarName, collector,
              benchmark, heapSize);
        }
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }

}

private void runIteration(String jarName,
```

```
                                String collector,
                                Entry<String, String> benchmark,
                                int heapSize)
      throws Exception {

    String garbageLogFile =
        String.format("Logs/g_%s_%s_%s.txt",
            collector, benchmark.getKey(), heapSize);
    String stopwatchFile = String.format("Logs/s_%s.txt",
        benchmark.getKey());

    runProcess(
        jarName,
        garbageLogFile,
        stopwatchFile,
        collector,
        benchmark,
        heapSize
    );

    if (Files.exists(Paths.get(stopwatchFile))) {
      Double garbageCollectionResult =
          getCollectionTime(garbageLogFile);
      Double executionTimeofAnIteration =
          getExecutionTime(stopwatchFile);

      writeResult(resultFile,
          String.format("%d\t%s\t%s\t%f\t%f",
              heapSize,
              benchmark,
              collector,
              garbageCollectionResult,
              executionTimeofAnIteration));

      Files.deleteIfExists(Paths.get(stopwatchFile));
    } else {
      writeResult("error.txt",
          String.format("failed␣benchmark␣%s␣" +
                  "with␣collector␣%s␣and␣heap␣%d",
              benchmark, collector, heapSize));
    }
  }

  public void runProcess(
      String jarName,
      String garbageLogFile,
      String stopWatchLogFile,
      String collector,
```

```
    Entry<String, String> benchmark,
    int heapSize
) throws Exception {

  Thread.sleep(1000);

  ArrayList<String> command = new ArrayList<>();

  command.add("java");
  command.add(String.format("-Xms%dm", heapSize));
  command.add(String.format("-Xmx%dm", heapSize));
  command.add(String.format("-Xloggc:%s", garbageLogFile));
  command.add("-XX:+PrintGCApplicationStoppedTime");
  command.add(String.format("-XX:+%s", collector));
  command.add("-jar");
  command.add(jarName);
  command.add(benchmark.getKey());
  command.add("-s");
  command.add(benchmark.getValue());
  command.add("--callback");
  command.add("CustomCallback");
  command.add("-C");
  command.add("-n");
  command.add("20");

  ProcessBuilder processBuilder =
      new ProcessBuilder(command);

  processBuilder.
      redirectOutput(ProcessBuilder.Redirect.INHERIT);

  processBuilder.
      redirectError(ProcessBuilder.Redirect.INHERIT);

  Process process = processBuilder.start();
  process.waitFor();
}


public double getCollectionTime(String garbageLogFile)
    throws Exception {

  double pauseTime = 0;

  ArrayList<String> lines =
      readAllLines(garbageLogFile);

  for (String line : lines) {
```

```
      if (line.contains("System.gc()")) {
        pauseTime = 0;
      }

      if (line.contains("Total␣time␣for␣which␣" +
          "application␣threads␣were␣stopped")) {
        String[] parts = line.split("stopped:␣");
        pauseTime +=
            Double.parseDouble(parts[1].substring(0, 9));
      }
    }

    return pauseTime;
  }

  public Double getExecutionTime(String stopWatchLogFile)
      throws Exception {
    Double executionTimeofAnIteration = 0.0;
    ArrayList<String> lines =
        readAllLines(stopWatchLogFile);

    executionTimeofAnIteration +=
        Double.parseDouble(lines.get(0).trim());

    return executionTimeofAnIteration / 1000;
  }

  public ArrayList<String> readAllLines(String path)
      throws Exception {

    ArrayList<String> lines = new ArrayList<String>();

    BufferedReader bufferedReader =
        new BufferedReader(new FileReader(path));
    String line;
    while ((line = bufferedReader.readLine()) != null) {
      lines.add(line);
    }

    bufferedReader.close();
    return lines;
  }

  public void writeResult(String file, String line)
      throws IOException {
    final Path path = Paths.get(file);
    Files.write(path, Arrays.asList(line),
```

```
            StandardCharsets.UTF_8,
            Files.exists(path) ?
                StandardOpenOption.APPEND :
                StandardOpenOption.CREATE);

    }
}
```

CustomCallback.java

```
import org.dacapo.harness.Callback;
import org.dacapo.harness.CommandLineArgs;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.Arrays;

public class CustomCallback extends Callback {

  public CustomCallback(CommandLineArgs args) {
    super(args);
  }

  public void start(String benchmark) {
    super.start(benchmark);
  }

  public void stop() {
    super.stop();
  }

  public void complete(String benchmark, boolean valid) {
    super.complete(benchmark, valid);

    if (valid && !this.isWarmup()) {
      try {
        Files.createDirectories(Paths.get("Logs"));
        writeResult(
            String.format("Logs/s_%s.txt", benchmark),
            String.format("%d", this.elapsed));
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
```

```
        System.err.flush();
    }

    public void writeResult(String file, String line)
        throws IOException {

        final Path path = Paths.get(file);
        Files.write(path, Arrays.asList(line),
            StandardCharsets.UTF_8,
            Files.exists(path) ?
                StandardOpenOption.APPEND :
                StandardOpenOption.CREATE);

    }
}
```

APPENDIX B

Table B.1 : Total execution time (seconds) results with 1024MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	28.831	25.388	33.989	21.571	
fop	0.447	0.523	0.414	0.418	0.473
h2	529.738	940.183			
jython	26.221	29.795	26.531	25.931	26.14
luindex	1.132	1.159	1.111	1.139	1.168
lusearch	3.669	3.721	3.512	3.533	3.521
pmd	3.004	3.507	3.11	3.168	2.918
sunflow	13.595	13.067	13.484	14.438	12.971
tradebeans	162.964	175.489	155.073	168.114	160.842
tradesoap	186.134	203.028	171.264	169.088	191.37
xalan	13.11	13.623	13.019	12.528	12.516

Table B.2 : Gc pause time (seconds) results with 1024MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	0.175168	0.174048	0.032099	0.046502	
fop	0.096174	0.121082	0.045461	0.061293	0.082692
h2	107.494773	520.811653			
jython	0.565473	0.39886	0.221803	0.188082	0.306799
luindex	0.031552	0.044443	0.028489	0.019603	0.017129
lusearch	0.437175	0.188171	0.244379	0.302128	0.273647
pmd	0.25111	0.26056	0.161469	0.218682	0.300064
sunflow	0.40509	0.354638	0.28108	0.243993	0.247679
tradebeans	6.957912	8.915576	5.448508	7.052911	12.724363
tradesoap	41.202768	42.011615	28.532629	27.949405	47.377788
xalan	0.476697	0.293924	0.334519	0.272081	0.375485

Table B.3 : Total execution time (seconds) results with 2048MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	36.008	37.65	35.734	38.248	37.23
fop	0.556	0.646	0.6	0.552	0.567
h2	448.827	493.429	413.819	413.374	465.468
jython	27.413	31.666	27.204	27.62	27.035
luindex	1.282	1.34	1.311	1.283	1.413
lusearch	4.368	4.065	3.69	3.698	3.698
pmd	3.377	4.565	3.206	3.331	3.291
sunflow	14.396	13.157	13.813	13.84	13.377
tradebeans	186.689	207.211	186.949	182.179	188.926
tradesoap	208.856	229.336	185.337	189.588	210.539
xalan	15.958	16.627	17.299	15.468	15.787

Table B.4 : Gc pause time (seconds) results with 2048MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	0.137813	0.038583	0.019059	0.022598	0.164689
fop	0.093345	0.115465	0.051616	0.043055	0.087582
h2	12.119171	45.570164	8.023844	10.941907	19.898319
jython	0.684131	0.582274	0.162831	0.172896	0.361053
luindex	0.036342	0.051404	0.027963	0.027525	0.026687
lusearch	0.516162	0.169325	0.143235	0.144673	0.124103
pmd	0.219498	0.492345	0.105768	0.084145	0.236929
sunflow	0.379606	0.489014	0.080532	0.123323	0.113298
tradebeans	7.698818	8.372097	6.142741	8.293813	12.348965
tradesoap	44.441505	43.049317	26.532947	26.63363	47.512897
xalan	0.585609	0.374548	0.183545	0.165159	0.185595

Table B.5 : Total execution time (seconds) results with 3072MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	36.725	39.522	37.574	37.129	38.268
fop	0.573	0.765	0.576	0.565	0.551
h2	457.414	482.371	403.652	453.568	479.104
jython	28.209	31.974	27.787	27.857	27.942
luindex	1.358	1.341	1.357	1.31	1.365
lusearch	4.283	3.966	3.731	3.743	3.601
pmd	3.328	4.936	3.4	3.26	3.324
sunflow	14.396	13.076	13.623	13.638	14.031
tradebeans	178.301	212.387	196.885	194.275	193.3
tradesoap	210.241	231.568	187.017	187.464	208.354
xalan	16.191	16.997	15.463	16.124	15.892

Table B.6 : Gc pause time (seconds) results with 3072MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	0.145061	0.107926	0.104707	0.098225	0.161922
fop	0.100284	0.131786	0.032698	0.031388	0.087193
h2	13.814161	28.47417	5.288093	4.631614	13.930134
jython	1.068836	0.551651	0.170294	0.167506	0.343654
luindex	0.041664	0.060803	0.03011	0.0306	0.023901
lusearch	0.74194	0.209924	0.08244	0.096194	0.133772
pmd	0.225834	0.533816	0.087217	0.126421	0.353532
sunflow	0.575418	0.293025	0.092108	0.085491	0.200664
tradebeans	8.361659	6.463944	7.825236	7.117908	11.652059
tradesoap	45.562661	43.679656	25.472553	25.679796	45.445509
xalan	0.774991	0.220788	0.156326	0.159512	0.247904

Table B.7 : Total execution time (seconds) results with 4096MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	22.201	31.721	30.474	22.841	32.209
fop	0.554	0.494	0.539	0.459	0.44
h2	410.68	396.344	374.378	364.038	392.981
jython	27.343	29.614	26.374	25.995	26.403
luindex	1.204	1.224	1.102	1.142	1.148
lusearch	4.446	3.543	3.345	3.273	3.343
pmd	2.746	2.835	2.747	2.847	3.576
sunflow	14.449	12.945	12.98	13.248	13.018
tradebeans	163.702	171.423	162.462	166.836	174.698
tradesoap	191.974	200.642	167.819	168.02	188.261
xalan	13.559	13.213	13.265	12.427	12.732

Table B.8 : Gc pause time (seconds) results with 4096MB heap size.

Benchmark	CMS GC	G1 GC	Par GC	ParOld GC	Ser GC
avrora	0.193537	0.151613	0.038209	0.113217	0.166713
fop	0.125719	0.133827	0.037265	0.036659	0.094412
h2	12.543741	13.818185	4.555378	4.661024	13.649065
jython	1.429874	1.119969	0.171112	0.182799	0.560238
luindex	0.064617	0.066339	0.04086	0.028405	0.022893
lusearch	1.094913	0.17456	0.121215	0.117771	0.136944
pmd	0.301882	0.523958	0.187218	0.170513	0.480343
sunflow	0.83114	0.533678	0.081801	0.135459	0.283993
tradebeans	9.040594	5.648511	5.490433	5.767771	10.285154
tradesoap	45.431507	37.934698	25.130972	25.31897	43.726402
xalan	1.0835	0.765129	0.155419	0.13238	0.324732

APPENDIX C

Allocation overview of all benchmarks are shared. First 25 classes which had most instances allocated are listed for each benchmark.

Class names are shared with their respective packages using classic java naming convention.

Table C.1 : Object allocation distribution of avrora benchmark.

Class Address	Allocated Object Count
java.util.LinkedList$ListItr	3705131
char[]	1988172
java.lang.String	998193
java.util.LinkedList$Node	886386
java.util.LinkedList	843500
avrora.sim.clock.RippleSynchronizer$WaitLink	161484
avrora.sim.clock.DeltaQueue$EventList	108638
java.nio.HeapCharBuffer	47600
java.lang.StringBuffer	46856
java.util.HashMap$Node	12178
byte[]	8372
java.lang.StringBuilder	7970
avrora.sim.AtmelInterpreter$IORegBehavior	7774
avrora.sim.util.MulticastFSMProbe	7774
avrora.sim.FiniteStateMachine$State	7514
java.lang.Integer	6664
avrora.sim.RWRegister	6396
avrora.arch.legacy.LegacyInstr$LDI	5531
avrora.arch.legacy.LegacyInstr$MOVW	4692
avrora.arch.legacy.LegacyInstr$STS	4219
avrora.arch.legacy.LegacyInstr$LDS	3659
avrora.arch.legacy.LegacyInstr$MULS	2952
cck.elf.ELFSymbolTable$Entry	2661
avrora.arch.legacy.LegacyInstr$CPC	2645
avrora.arch.legacy.LegacyInstr$RJMP	2508
Others	89561
Total	8969030

Table C.2 : Object allocation distribution of fop benchmark.

Class Address	Allocated Object Count
char[]	157800
java.lang.String	101538
org.apache.fop.fo.FObj$FObjIterator	92651
byte[]	21724
java.lang.StringBuffer	19006
org.apache.fop.traits.MinOptMax	17133
java.lang.Object[]	16562
java.util.LinkedList$Node	15862
org.apache.fop.fo.properties.CondLengthProperty	14384
org.apache.fop.fo.properties.SpaceProperty	10571
java.lang.Integer	10388
org.apache.fop.layoutmgr.LeafPosition	9748
java.util.HashMap$Node	9574
org.apache.fop.fo.properties.Property[]	9507
java.util.ArrayList	9486
org.apache.fop.fo.properties.EnumProperty	8995
java.util.LinkedList	8384
org.apache.fop.datatypes.LengthBase	8124
java.lang.Double	7745
org.apache.fop.fo.expr.PropertyInfo	7376
org.apache.fop.fo.expr.PropertyParser	7376
org.xml.sax.helpers.LocatorImpl	6817
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	5764
org.apache.fop.fo.properties.EnumNumber	5755
org.apache.fop.fo.properties.FixedLength	5749
Others	195865
Total	783884

Table C.3 : Object allocation distribution of h2 benchmark.

Class Address	Allocated Object Count
char[]	144055687
java.lang.String	98341531
org.h2.value.Value[]	48889809
org.h2.util.ObjectArray$ObjectArrayIterator	48244886
org.h2.index.TreeCursor	40938152
int[]	31582849
org.h2.result.Row	30888119
java.lang.StringBuilder	27181179
java.util.HashMap$Node	26503362
java.math.BigDecimal	21153475
java.math.BigInteger	17994800
org.h2.index.TreeNode	15336043
org.h2.expression.ExpressionVisitor	13979895
org.h2.value.ValueShort	13597443
java.math.MutableBigInteger	12874853
org.h2.index.MultiVersionCursor	11162026
java.util.HashMap	10879803
java.util.HashSet	9867934
org.h2.value.ValueString	8866906
java.lang.Object[]	8151840
org.h2.util.ObjectArray	8010518
org.h2.value.ValueDecimal	6950666
java.lang.Integer	6488796
org.h2.expression.Expression[]	6475452
org.h2.result.LocalResult	6474822
Others	40505574
Total	715396420

Table C.4 : Object allocation distribution of jhyton benchmark

Class Address	Allocated Object Count
org.python.core.PyObject[]	24790090
org.python.core.PySequence$1	22319150
char[]	19786959
java.lang.String	17155525
org.python.core.PyString	11917810
org.python.core.PyFrame	10491531
int[]	9675077
org.python.core.PySlice	7059528
java.util.concurrent.ConcurrentHashMap$Node	6205147
org.python.core.PyUnicode	4895026
org.python.core.PyFloat	4563235
org.python.core.PyComplex	4000019
org.python.core.PyTuple	3819951
java.lang.String[]	3324430
java.util.HashMap	3263822
java.util.HashSet	3263011
org.python.core.ArgParser	3261133
org.python.core.PyInteger	3217879
java.util.ArrayList	2876544
java.util.concurrent.ConcurrentHashMap	2388288
org.antlr.runtime.tree.RewriteRuleSubtreeStream	2364624
org.python.core.PyDictionary$dict_get_exposer	2300024
org.python.core.PyLong	2281815
java.math.MutableBigInteger	2161269
org.antlr.runtime.tree.RewriteRuleTokenStream	1972200
Others	32359040
Total	211713127

Table C.5 : Object allocation distribution of luindex benchmark.

Class Address	Allocated Object Count
char[]	68733
java.lang.String	53546
org.apache.lucene.index.TermInfo	35643
org.apache.lucene.index.Term	35629
org.apache.lucene.index. FreqProxTermsWriter$PostingList	21504
byte[]	8464
int[]	8212
java.util.HashMap$Node	6404
java.lang.Long	6315
java.nio.HeapCharBuffer	5209
java.util.Date	4943
org.apache.lucene.document.Field	3702
java.io.File	3376
java.lang.StringBuffer	3110
java.lang.ref.Finalizer	3057
java.lang.Object	3019
java.io.FileDescriptor	2566
boolean[]	2475
java.util.Calendar$Builder	2475
java.util.GregorianCalendar	2475
sun.util.calendar.Gregorian$Date	2475
java.util.zip.ZipEntry	2466
java.nio.file.attribute.FileTime	2187
java.lang.StringBuilder	2091
java.util.ArrayList	1485
Others	26640
Total	318201

Table C.6 : Object allocation distribution of lusearch benchmark.

Class Address	Allocated Object Count
char[]	1661294
java.lang.String	987716
int[]	526042
org.apache.lucene.index.Term	407533
org.apache.lucene.queryParser.Token	393205
java.util.ArrayList	316266
java.lang.Object[]	271576
java.io.StringReader	262190
org.apache.lucene.search.TopDocCollector	262183
org.apache.lucene.search.HitQueue	262177
org.apache.lucene.search.TopDocs	262173
org.apache.lucene.search.ScoreDoc[]	262167
float[]	262145
org.apache.lucene.search.TermScorer	262111
org.apache.lucene.index.SegmentTermDocs	262097
org.apache.lucene.search. TermQuery$TermWeight	262097
java.io.IOException	262090
org.apache.lucene.queryParser.FastCharStream	131166
org.apache.lucene.queryParser. QueryParser$JJCalls	131162
org.apache.lucene.analysis.Token	131096
org.apache.lucene.analysis.StopFilter	131094
org.apache.lucene.analysis.LowerCaseFilter	131091
org.apache.lucene.analysis.standard. StandardTokenizer	131090
org.apache.lucene.analysis.standard. StandardTokenizerImpl	131090
org.apache.lucene.analysis.standard. StandardFilter	131089
Others	688546
Total	8922486

Table C.7 : Object allocation distribution of pmd benchmark.

Class Address	Allocated Object Count
char[]	1796032
java.lang.String	1211604
java.util.ArrayList	992665
net.sourceforge.pmd.ast.Token	831742
byte[]	677965
java.lang.Object[]	584968
org.jaxen.Context	444966
net.sourceforge.pmd.jaxen.Attribute	417271
net.sourceforge.pmd.ast.Node[]	378575
java.util.ArrayList$Itr	378448
java.util.HashMap$Node	367137
java.util.HashMap	329266
java.util.HashMap$Node[]	308095
java.util.LinkedList$ListItr	305729
java.util.HashSet	300516
java.util.HashMap$KeyIterator	229505
org.jaxen.expr.IdentitySet$IdentityWrapper	217377
net.sourceforge.pmd.symboltable. ImageFinderFunction	199584
java.lang.Integer	170883
org.jaxen.util.SelfAxisIterator	159243
net.sourceforge.pmd.jaxen. DocumentNavigator$1	148956
java.util.AbstractList$Itr	127515
org.jaxen.util.SingletonList	127235
org.jaxen.expr.IdentitySet	99181
java.util.LinkedList$Node	94308
Others	2401322
Total	13300088

Table C.8 : Object allocation distribution of sunflow benchmark.

Class Address	Allocated Object Count
org.sunflow.core.Ray	84794015
org.sunflow.image.Color	22058874
org.sunflow.math.Vector3	10707476
double[]	3372990
org.sunflow.core.LightSample	2664026
org.sunflow.math.Point3	2184336
org.sunflow.core.renderer. BucketRenderer$ImageSample	809367
byte[]	268272
org.sunflow.math.OrthoNormalBasis	261234
org.sunflow.math.Point2	261138
org.sunflow.core.ShadingState	261126
org.sunflow.core. ShadingState$LightSampleIterator	210693
float[]	26957
char[]	25032
java.lang.String	14952
int[]	13322
long[]	6292
org.sunflow.core.accel.KDTree$BuildTask	6280
java.util.HashMap$Node	5927
org.sunflow.core.primitive. TriangleMesh$WaldTriangle	4108
java.lang.StringBuilder	2956
jdk.internal.org.objectweb.asm.Item	2555
java.nio.file.attribute.FileTime	2187
java.lang.Object[]	2135
org.sunflow.core.IntersectionState$StackNode	2037
Others	41763
Total	128010050

Table C.9 : Object allocation distribution of tradebeans benchmark.

Class Address	Allocated Object Count
char[]	253381381
java.lang.String	160233545
java.util.HashMap$Node	75034905
org.h2.util.ObjectArray$ObjectArrayIterator	57204991
java.lang.Object[]	47290394
org.h2.util.ObjectArray	39717131
java.lang.StringBuilder	33600998
org.h2.expression.ExpressionColumn	31122027
org.h2.index.TreeCursor	28345554
java.lang.Long	26139472
org.h2.value.Value[]	24022089
org.h2.expression.ExpressionVisitor	21444938
java.math.BigDecimal	14816624
int[]	14214455
org.h2.index.TreeNode	12599862
org.h2.result.Row	12104379
java.util.HashMap	11777856
org.h2.expression.Expression[]	11654897
org.h2.expression.Parameter	11476390
java.util.HashMap$Node[]	11384362
org.h2.table.TableFilter[]	10291517
org.h2.result.LocalResult	8205971
org.h2.expression.Wildcard	6035542
java.util.HashSet	5309260
org.h2.command.Parser	4878292
Others	115750322
Total	1048037154

Table C.10 : Object allocation distribution of tradesoap benchmark.

Class Address	Allocated Object Count
char[]	228001434
java.lang.String	127163192
java.util.HashMap$Node	50582921
java.lang.StringBuffer	29557029
java.lang.reflect.Field	29327487
java.lang.Object[]	27355983
java.util.ArrayList	27270370
javax.xml.namespace.QName	22996790
com.sun.org.apache.xerces.internal.util. PropertyState	22303132
com.sun.org.apache.xerces.internal.util. FeatureState	19986834
org.apache.axis.encoding. TypeMappingImpl$Pair	18556180
org.apache.axis.utils.Mapping	18516365
org.h2.util.ObjectArray$ObjectArrayIterator	14339568
java.util.HashMap	12782598
java.lang.StringBuilder	11522918
java.util.ArrayList$Itr	11441264
java.lang.Long	10037100
java.io.CharArrayWriter	10010248
org.h2.util.ObjectArray	9963056
org.apache.axis.utils.BeanPropertyDescriptor	9523923
byte[]	8824460
java.util.HashMap$Node[]	8538557
org.h2.expression.ExpressionColumn	7809128
int[]	7792330
java.util.Vector	7505097
Others	246015332
Total	997723296

Table C.11 : Object allocation distribution of xalan benchmark.

Class Address	Allocated Object Count
java.nio.HeapCharBuffer	27671441
char[]	24607639
java.util.HashMap$Node	4562914
java.lang.String	2633212
org.apache.xpath.objects.XNodeSet	2537457
int[]	808275
org.apache.xml.dtm.ref. ExpandedNameTable$HashEntry	715106
byte[]	402687
org.apache.xpath.NodeSetDTM	371904
org.apache.xml.serializer.ElemContext	371207
java.lang.StringBuffer	367788
java.lang.Object[]	350169
org.apache.xpath.objects.XString	282043
java.lang.StringBuilder	277165
org.apache.xml.dtm.ref.ExtendedType	269976
java.util.Hashtable$Entry	234060
org.apache.xpath.axes. NodeSequence$IteratorCache	228963
org.apache.xml.dtm.ref. DTMDefaultBaseIterators$ParentIterator	214802
java.util.StringTokenizer	186961
java.lang.String[]	155837
boolean[]	148420
java.io.File	125824
java.util.Hashtable$Entry[]	125804
int[][]	125573
char[][]	122225
Others	4043832
Total	71941284

CURRICULUM VITAE

Name Surname: Gencay Sazak

Place and Date of Birth: Mustafakemalpaşa - 26 February 1990

E-Mail: sazakg@gmail.com

EDUCATION:

- **B.Sc.:** 2012, Istanbul Technical University, Faculty of Computer and Informatics Engineering, Computer Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2011 - 2016 - VeriPark - Software Developer
- 2016 - ... - VeriPark - Team Leader

PRESENTATIONS ON THE THESIS

Sazak G., Ovatman T., 2017: Java Garbage Collector Performance on Multicore: The Effect of Data Structures and Algorithms. *Accepted and presented in UBMK 2017, 2017 International Conference on Computer Science and Engineering (UBMK)*, 5-8 Oct. 2017, Antalya, Turkey