Proceedings of the 53rd Hawaii International Conference on System Sciences | 2020

Adaptive and Concurrent Garbage Collection for Virtual Machines

Md Enamul Haque, SM Zobaed, Razin Farhan Hussain, Aminul Islam, and Christian Traylor
School of Computing & Informatics
University of Louisiana at Lafayette
LA 70503, USA
{enamul, sm.zobaed1, razin-farhan.hussain1, aminul, christiantraylor1}@louisiana.edu

Abstract

An important issue for concurrent garbage collection in virtual machines (VM) is to identify which garbage collector (GC) to use during the collection process. For instance, Java program execution times differ greatly based on the employed GC. It has not been possible to identify the optimal GC algorithms for a specific program before exhaustively profiling the execution times for all available GC algorithms. In this paper, we present an adaptive and concurrent garbage collection (`ACGC`*) technique that can predict the optimal GC algorithm for a program without going through all the GC algorithms. We implement this technique in the Java virtual machine and test it using standard benchmark suites.* `ACGC` *learns the algorithms' usage pattern from different training program features and generates a model for future programs. Feature generation and selection are two important steps of our technique, which creates different attributes to use in the learning step. Our experimental evaluation shows improvement in selecting the best GC. Additionally, our approach is helpful in finding better heap size settings for improved program execution.*

1. Introduction

Memory allocation and program execution are two significant and related aspects in the overall operating systems performance. This relationship is defined using the concept of a working set of an application, which refers to a set of currently running objects of that particular application [1]. The working set size is defined by the amount of memory required to store those objects. When the application's working set size exceeds available memory, throughput becomes limited by waiting for memory to be paged in or out. Hence, the program performance can be improved by increasing the memory size, which in turn reduces the number of page faults. However, the application stops paging when its working set fits into main memory. In that scenario, performance might not improve with the increase in memory capacity. To overcome such situations, garbage collection is a technique that reclaims dynamically allocated memory locations, helps developers from freeing the memory explicitly in the program.

One of the prime features of managed runtime environment is automatic memory management through garbage collection. In managed runtime environment such as Java Virtual Machines (JVM) and the Common Language Runtime (CLR), the total execution time of an application depends on both application and VM execution [2]. The most significant time-consuming tasks performed by the VM is reclaiming the memory usage and garbage collection. In this paper, we focus on time reduction by GC which eventually reduces execution time of an application.

GC has a dominating effect on the net execution time of an application for two reasons. Firstly, a significant portion of the net execution time is spent for garbage collection. Experimental results show that the mean proportion of execution time spent in GC is 12.2% for 1566 experimental configurations [2]. Secondly, an indirect impact is caused by the method in which GC algorithm rearranges heap-allocated data after collection [3]. Therefore, it affects latter program execution time due to changes in spatial locality of data.

There exists different garbage collection techniques that efficiently manage memory allocation and de-allocation for applications with varying resource requirements. Additionally, it is well-known that different GCs perform better for different programs. Hence, selecting the best GC algorithm for every program is a challenging task that requires either exhaustive profiling of all the applications on all available GCs or learning from program specific historical GC usage pattern. In this paper, we show wasted CPU utilization in the current virtual machine. Additionally, we present a new program specific to adaptive and concurrent GC (`ACGC`) selection technique that considerably reduces GC CPU utilization (percentage of CPU time spent in GC) and improves scalability and GC performance.

This paper makes the following contributions:

- We introduce an adaptive and concurrent garbage collection scheme that can be used on top

URI: https://hdl.handle.net/10125/63950
978-0-9981331-3-3

of existing virtual machine to achieve efficient execution of programs.

- We use non-negative matrix factorization (NNMF) for feature transformation that improves classification accuracy in optimal GC selection.
- We analyze the effects of heap size setting (initial, minimum, and maximum) for different benchmark data to exploit the garbage collection behavior of different GCs.
- We use four different features from benchmark programs to analyze the optimal collection.
- Our experimental results demonstrate promising improvement over traditional features used for garbage collection analysis.

The rest of the paper is organized as follows: Section 2 presents a comprehensive literature review of application specific garbage collection schemes. Section 3 presents background and problem statement of the proposed approach. Section 4 describes the solution using pattern learning algorithm from program usage behavior. Section 5 and 6 presents the experimantal settings and the empirical evaluation of our technique respectively. Finally, Section 7 concludes our work.

2. Background

The load balancing mechanisms of different single and multi-threaded GC have been presented in the literature. A novel JVM GC approach is proposed in [4] where efficient GC performance, reduction of GC's CPU usage, and scalability were claimed after analyzing Google data-center jobs and DaCapo benchmark suite.

Data Structure Aware (DSA) interface enabled GC has been proposed by Cohen et al. [5]. GC gets information through the DSA to handle such data structures which are often used to hold much of the program data. The authors claimed that the proposed method reduces the overall program execution time. Additionally, GC's performance is enhanced by utilizing the knowledge of the most frequently used data structures such as arrays and linked lists. However, the DSA interface is not scalable to all programs and also not capable to work on major data structure with significant amount of data.

In the presence of automatic memory management, the relationship has been found between allocated memory and GC, indicating application performance. Larger heap sized memory reduces the frequency of GC operation. But once the heap size exceeds the available physical memory, parts of the heap is paged to the backing store. Zhang et al. [1] proposed a scheme for adaptively identifying the optimal heap size for a program while it is running within a Java virtual machine. However, the efficiency of GC performance does not fully depend on optimal heap size.

Andreasson et al. applied ML methods to enhance the GCs for more adaptive solutions [6]. They used reinforcement learning, in which an agent interacts with the environment and learns by trial and error rather than from direct training examples. This work analyzed an adaptive decision process that makes decisions regarding which GC technique should be invoked and how it should be applied. The report has investigated how to design and implement a learning decision process for a more dynamic garbage collection in a modern JVM. The result of their analysis is that the use of a reinforcement learning system is particularly useful if an application has a complex dynamic memory allocation behavior. For the above reason, the dynamic garbage collection technique is proposed at first place in this research work.

Blackburn et al. [7] presented design, implementation, and evaluation of a memory management toolkit (MMTk) for java that is used to develop the java GC. Most of the existing GCs are monolithic and do not share reused components whereas MMTk use these two features. This paper is a case study that shows flexibility can actually improve rather than degrade performance. Here they showed how MMTk combines good software engineering design with excellent performance by comparing MMTk code and execution times in Jikes RVM, a Java-in-Java Virtual Machine, with monolithic Java and C collector implementations.

Bruno et al. [8] presented a comprehensive study on various GC algorithms that are utilized in Big Data [9, 10] and machine learning platforms [11, 12]. They claimed that by analyzing memory profiling, rate of successful collection, and maintaining a balanced heap size, classic GC algorithms can be improved especially the scalability issue. The study mainly focuses on the improvement of GC rather than identifying suitable program specific GC.

In Table 1, we provide a summary of advantages and drawbacks of the prior works. Unlike previous approaches, our paper presents a novel method that uses both matrix factorization and machine learning to efficiently select an optimum GC algorithm for a program that has not been seen before by the profiler.

3. Problem Statement

We consider the best GC selection for a particular program as an optimization problem. In our problem setting, we assume that the system has $C = \{c_1, c_2, \ldots, c_n\}$ GCs where c_i refers to i-th GC. Additionally, we assume that the system will execute $P = \{p_1, p_2, \ldots, p_m\}$ programs where p_j denotes j-th program. Without the loss of generality, let us assume

Table 1: **A summary of the studied works**

Works	Advantages	Drawbacks
[5]	Performance enhancement.	Supported limited data structure.
		Limited scalability.
		Limited compatibility.
		No discussion on CPU usage.
		No suggestions of GC.
[1]	Identified optimal heap size for a java program.	Depends on other params.
		Not scalable.
		Only compatible with Java
[6]	Applied ML to enhance the GC techniques.	No prediction was provided for program specific appropriate GC.
		Discussed only the improvement in the decision process of GCs.
		No suggestions of GC.
[13]	Offered a portable toolkit to develop GC.	Only improves the architecture of monolithic java, and C GC.
		No suggestions of GC.

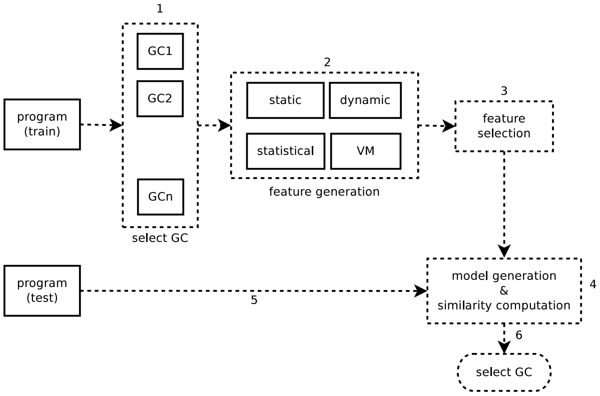


Figure 1: **Process diagram of adaptive and concurrent GC (ACGC).**

that *net execution time* of an application depends on the type of a program and individual GC. Note that, *net execution time* consists of both *execution time* and *garbage collection time*. Considering *execution time* ($\mathcal{E}(p_j, c_i)$) and *garbage collection time* ($\mathcal{G}(p_j, c_i)$) as functions of program and garbage collection, we present our optimization function in Equation 1 that is solved using learning algorithms (Section 5).

$$\arg\min_{c_i} \big[\mathcal{E}(p_j, c_i) + \mathcal{G}(p_j, c_i)\big] \tag{1}$$

$$\arg\min_{c_i} \mathcal{G}(p_j, c_i) \tag{2}$$

As execution time primarily depends on the processor configuration, we specifically focus on the garbage collection time. Hence, we can rewrite Equation 1 using only garbage collection function in Equation 2. We assume that the execution time remains same for a particular program.

We now present a toy example to make a simple and concrete scenario of the above problem statement. Suppose we know (after experimentation) that a program P executes in the shortest time using GC algorithm `gc_hello`, and program Q executes in the shortest time using GC algorithm `gc_world`. We want to know which GC algorithm works best in terms of shorter execution time for a new program R without profiling all the available GC techniques in the system. Our goal is to avoid the profiling complexity by introducing adaptive and concurrent selection method.

4. Proposed Approach

In this section, we present our solution approach of the above mentioned problem statement using Figure 1. Firstly, we measure execution time for a particular program by profiling all the available GCs. At this stage, all the programs make up the training dataset. The GC that achieves minimum execution time is set as default label for that program. In the next step, each program is run using four different functions that extract features: i) static metrics such as depth of inheritance tree (DIT), ii) dynamic metrics such as number of allocated objects, iii) statistical metrics such as entropy, and iv) virtual machine (VM) metrics such as heap size. In the third step, we transform the feature space into latent space using NNMF to ensure similarity computation in the later stages by Equation 3,

$$\arg\min_{X^l, Y^l > 0} \frac{1}{2} ||X - X^l Y^l||_F^2 \tag{3}$$

where feature matrix X is transformed into a l dimensional feature space with two components X^l and Y^l such that the approximation in Equation 4 holds.

$$X \approx X^l Y^l \tag{4}$$

We also build a learning model using original feature space to adapt the two approaches for GC selection.

In the ***first approach*** (`ACGC+cosine`), we compute similarity score between test and training program features. The GC algorithm for the test program is set as default label by taking majority vote from the similarity scores. The approach is described in Algorithm 1.

In the ***second approach*** (`ACGC+learners`), we use a learning function $\mathcal{F}$, that maps the test program to a optimum GC algorithm. Let $\mathcal{X}$ denote the feature set of test program (benchmark) instances and let $\mathcal{Y} = \{1, 2, \ldots, Q\}$ be the finite set of labels, which corresponds to unique garbage collection algorithms. Given a training program set $T = \{(x_1, Y_1), (x_2, Y_2), \ldots, (x_m, Y_m)\}$ $(x_i \in X, Y_i \in Y)$, the goal of the learning system is to output a multi-class classifier $h : \mathcal{X} \to \mathcal{Y}$ which optimizes some evaluation metric. The approach is described in Algorithm 2.

5. Experimental Settings

Our purpose is to take several measurement aka *features* from those. A learning algorithm takes

Algorithm 1: ACGC using non-negative matrix factorization as feature generation with cosine similarity.

```
Input: Training data $\mathcal{T} = \{\mathcal{X}^{train}, \mathcal{Y}\}$,
Test data $\mathcal{D} = \mathcal{X}^{test}$
Set: Latent factors, $l$, $\mathcal{Y}^{test} = \emptyset$
Output: Optimal GC set $\mathcal{Y}^{test}$
Training:
1. Factorize $\mathcal{X}^{train}$ into $l$ dimensional latent space
   to get $X^l$ and $Y^l$ components using Equation 3.
Test:
repeat
    1. Randomly select a test program feature
       $x^{test}$.
    2. Compute similarity score $\sigma_{test}$ from $x^{test}$
       to all the features in $X^l$ using cosine
       similarity as below
    $\sigma_{i,j}^{test} = \frac{\sum_{k=1}^{l} x_{ik}^{test} X_{jk}^{l}}{\sqrt{\sum_{k=1}^{l} x_{ik}^{test2}}\sqrt{\sum_{k=1}^{l} {X_{jk}^{l}}^{2}}}$
    3. Calculate majority vote to select $Y^{best}$
       using $\sigma_{test}$ for $x^{test}$.
    4. Set $\mathcal{Y}^{test} = \mathcal{Y}^{test} \cup Y^{best}$.
until $\mathcal{D} = \emptyset$;
return $\mathcal{Y}^{test}$
```

a program execution specific behaviors and predicts appropriate GC algorithm that performs the best for a given heap size, without running all available GC algorithms. To collect program execution behaviors, we acquire information based on four categories: i) static, ii) dynamic, iii) virtual machine, and iv) statistical metrics. These information are appended together for a specific program that represent an initial feature space to the learning algorithm. Later, we transform the initial feature space into a latent space to ensure bias-free and similarity based classification.

5.1. Feature Generation and Selection

5.1.1. Static Metrics Among several metrics suite we adopt the Chidamber and Kemerer java metrics (*ckjm*) suite for object-oriented programs [14–16]. This includes the following eight measurements for each class: depth of inheritance tree (DIT), weighted methods per class (WMC), number of children (NOC), coupling between object classes (CBO), response for a class (RFC), lack of cohesion methods (LCOM), afferent couplings (Ca), and number of public methods (NPM).

The depth of inheritance tree (DIT) is defined as the maximum path length from the root. Figure 2 shows an example where R is considered as the root class. In this scenario, A and B has DIT score of 2. Similarly C and D has DIT score of 1.

Algorithm 2: ACGC using non-negative matrix factorization as feature generation with learning algorithms.

```
Input: Training data $\mathcal{T} = \{\mathcal{X}^{train}, \mathcal{Y}\}$,
Test data $\mathcal{D} = \mathcal{X}^{test}$
Set: Latent factors, $l$, $\mathcal{Y}^{test} = \emptyset$
Output: Optimal GC set $\mathcal{Y}^{test}$
Training:
1. Factorize $\mathcal{X}^{train}$ into $l$ dimensional latent space
   to get $X^l$ and $Y^l$ components using Equation 3.
Test:
repeat
    1. Randomly select a test program feature
       $x^{test}$.
    2. Compute $Y^{best}$ using learning function.
    3. Set $\mathcal{Y}^{test} = \mathcal{Y}^{test} \cup Y^{best}$.
until $\mathcal{D} = \emptyset$;
return $\mathcal{Y}^{test}$
```

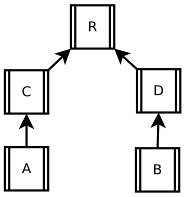


Figure 2: **A simple depiction of depth of inheritance tree with five classes where R refers to the root and others as inherited classes.**

Weighted methods per class (WMC) is the sum of the complexities of its methods. Complexity Among other measures of complexity, cyclomatic complexity [17] or arbitrarily assigned parameters can be used. The *ckjm* program assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in a class.

Number of children (NOC) metric is defined as the number of immediate subclasses subordinated under a class within a class hierarchy. This also denotes the total number of direct subclasses of current class.

The coupling between object classes (CBO) denotes total number of outside classes which are referred to from the current class. It measures how often a class uses instances from other classes. To remove ambiguity in computing CBO, multiple accesses to the same class from a specific class is counted as single access.

Response for a class (RFC) denotes a set of methods that can potentially be executed in response to a message received by that class. In other words, RFC denotes the total frequency of method execution for a single method call. In practice, we want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating the *transitive closure* [18] of the method's call graph. However, the process can be both expensive and inaccurate. In *ckjm*, a rough approximation is computed by simply inspecting method calls within the class's method bodies.

LCOM measures the variation of instance variable usage by different methods of a class. A lower LCOM score means most of the methods use a similar set of instance variables. Similarly, a higher score refers to disjoint set of instance variables usage.

Finally, CA and NPM scores for a class measure how many other classes use that specific class and the total number of public methods in the class respectively.

5.1.2. Dynamic Metrics on Reference VM We collect specific object demographics for each benchmark program by using standard JVM. ideally, those demographics reflect program allocation behavior. These metrics are: number of allocated objects, average of allocated objects' space, and the total execution time for a particular arrangement. In this case arrangement denotes which GC algorithm is applied based on initial, minimum, and maximum heap size (16M, 32M, 48M, ..., 512M). We collectively call the metrics as object or arrays which are GC independent. For instance, we do not consider nursery space as a member of the object array. Nursery space is GC dependent feature as it is only available when GenMS algorithm is called. Overall, we analyze that GC dependent dynamic metrics have minimal effect on performance.

Our goal is that one GC algorithm may outperform other for different sized objects or arrays. We measure GC performance based on program completion such as execution time. Note that, the allocation and collection of Large Object Space (LOS) are statistically independent to the general purpose JVM heap.

The above metrics, with all absolute values, provide an insight into the program's GC behavior with different heap size. In our next step, we consider the metrics as different features which are also responsible for improving program specific GC algorithm prediction accuracy along with other metrics.

5.1.3. VM Metrics Heap size is one of the influential metrics among the popular JVM metrics to be considered for garbage collection. Heap is the memory used by an application for creating and storing objects. Heap is where objects created by an application live. Run out of heap means application can no longer create new objects and it leads to "OutOfMemory" errors for running applications. This error can have very serious effects on the JVM. For example, it can stop the JVM, force a heap dump, pause the JVM, or it can kill the JVM. Garbage Collection is the mechanism that discards unused objects from the heap, reclaiming the space for application use. On the other hand, GC is a resource intensive process that leads to poor performance of a program. So depending on GC algorithm, programs have different performance throughput [19].

In this paper, we consider heap size and garbage collection techniques as input metrics. In the case of heap size, three different parameters are considered such as initial heap size, minimum heap size and maximum heap size. The mentioned parameters are specified to JVM by command line flags as `-Xms<Number>` (initial heap size), `-Xmn<Number>` (minimum heap size) and `-Xmx<Number>` (maximum heap size), respectively. In addition, five popular GC algorithms are used as input parameters which are Parallel GC, Concurrent Mark-Sweep GC(CMS), G1GC, Parallel New GC and Serial GC. The command line flags for using the mentioned GC's are presented in Table 2.

Table 2: **Command line flags for the five garbage collection algorithms.**

GC name	Command line flag
Parallel GC	`XX:+UseParallelGC`
Concurrent Mark Sweep	`XX:+UseConcMarkSweepGC`
G1 GC	`XX:+UseG1GC`
ParNew	`XX:+UseParNewGC`
Serial GC	`XX:+UseSerialGC`

We consider 11 values: 16, 32, 48, 64, 96, 128, 160, 192, 224, 256, and 512 respectively for the heap size attribute. The unit of these values are in megabytes. So, we consider permutation of the heap sizes for three different parameters and five GC algorithms to run on individual benchmark application. As heap size plays an important role while running different GC algorithms on a particular application, we consider this as a critical feature. Therefore there are 5×11^3 permutations of commands used for collecting execution time of benchmark applications which is considered as training data for the machine learning model. We have used 6655 sets of input parameters on each benchmark application and find the GC algorithm that provides minimum execution time for each particular application.

Singer *et al.* [2] used Jikes RVM which has the problem of booting if the heap size is less than 8MB. For that reason they considered the smallest heap size for which the application successfully completes and there is no out-of-memory errors from Jikes RVM compiler.To avoid the issue we overlooked using Jikes RVM and we set the lowest heap size to 16 megabytes. In a similar work, Soman *et al.* [20] used a single metric to decide when to change the GC algorithm. They used the ratio of the current heap size to the minimum possible heap size for each benchmark. `DaCapo` study represents separate heaps for VM objects and for application objects [13]. We use `DaCapo` as one of the benchmarks. As SPECjvm2008 [21] is targeted for measuring the performance of both JVM and hardware systems, we used this benchmark for training our model.

As our work is focused on performance of application with respect to GC allocation techniques, preference is given to find the GC technique which provides minimum execution time. For that reason heap size and GC allocation algorithms are taken as VM metrics for feature extraction.

5.1.4. Statistical Metrics Most of the works consider a subset of the earlier metrics as features. However, we find several motivations to use additional statistical metrics as features over the standalone use. For example, two different classes, with similar behavior such as method call, are likely to provide similar static metrics. Those classes might have other local and global variable declaration along with conditional loops and branches. To avoid having similar static metrics for such classes, we intend to apply statistical metrics such as entropy and complexity in parallel with other metrics as initial features. We use Shannon entropy to calculate the quantity of information from a discrete source such as class files of benchmark programs. Entropy is a measure of unpredictability or information content in a random variable. For a random variable X with n outcomes $x_1, \ldots, x_n$, the Shannon entropy, a measure of uncertainty is defined by $H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i)$.
We also use complexity of a program x that measures the length of the shortest program that generates x. Among different compression algorithms used for measuring complexity we used Lempel-ziv algorithm [22].

5.2. Benchmarks

In the following, we describe the benchmark programs used in our experiments. Note that lack of good benchmark suites is one of the significant issues in memory management research. While there is certainly more scope for the improvement, we select three widely used memory management benchmarks such as SPECJvm2008, Dacapo, and JOlden[1]. We briefly mention the benchmark information in Table 3.

The three benchmark suites include different types of programs such as mathematical computations, sorting, and image processing. The programs consists of simple to complex and time consuming computation. Hence, the GC algorithms encounter low to high load during the collection for these programs.

Table 3: **Benchmarks used for the experiments**

Suite	Benchmark	Description
JOlden	bh	Mathematical computation
	bisort	Sorting
	em3d	Mathematical computation
	health	Process simulation
	mst	Minimum spanning tree
	perimeter	Image processing
	power	Process simulation
	treeadd	Tree traversal
	tsp	Graph optimization
	voronoi	Image processing
SPECJvm2008	compiler	Front end Java compiler
	compress	Data compression
	crypto	Cryptographic algorithms
	derby	Database logic and lock computation
	MPEGaudio	Mp3 decoder
	scimark	Floating point computation
	serial	Serialization and de-serialization of objects
	startup	Multi-threaded optimization
	sunflow	Graphics visualization
	xml	XML transformation
Dacapo	avrora	AVR microcontrollers
	tomcat	Web server
	batik	Scalable Vector Graphics (SVG) images
	eclipse	JDT performance tests
	xalan	XML to HTML converter
	fop	PDF generator

5.3. GC Algorithms

In the following we present five different GC algorithms which are used in most of the Java virtual machines. Serial GC is mainly designed for single threaded environment and for simple heap. It uses mark-copy algorithm for the Young Generation and mark-sweep-compact for the Old Generation which are incapable of parallelizing the tasks. Young Generation means the location where most of the newly created objects are located. After the creation, some objects disappear from the location. When objects are removed from Young Generation, we say a minor GC is performed. Old Generation means the location where the set of objects that are reachable and also survived from the texttt Young Generation are located. The size of the Old Generation is bigger and generally performs GC when the allocated space for old generation is full and this is why GC occurs less frequently here in compare to texttt Young Generation. The GC stops all application threads whenever it is working. As the Serial GC does not use multiple cores, it is not recommended to use in server environment.

While the serial GC uses only one thread to perform garbage collection, the parallel GC uses several threads to process a GC. In addition, it performs minor collection in parallel which reduces garbage collection overhead [2]. This GC is helpful when there is sufficient main memory and a large number of cores. It is also called the "throughput GC". However, these collectors are still prone to long pauses and in that time interval, application threads are stopped.

Like parallel GC, the G1 collector is a server-side GC, designed for multi-processor machines with enough memories. G1 GC is built to make the duration of *stop-the-world* pauses because of garbage collection predictable and also, configurable. It is known that G1

[1] http://www.mm-net.org.uk/resources/benchmarks.html

[2] https://codeahoy.com/2017/08/06/basics-of-java-garbage-collection/

is a soft real-time GC which means programmers are allowed to set certain performance goals.

The CMS algorithm is built to avoid long pauses due to performing collection at the `Old Generation`. The algorithm has two stages. At the beginning, instead of compacting the `Old Generation`, it uses free lists to manage reclaimed space. It mostly performs collection activity in the mark-and-sweep phases concurrently with application. The aim of doing that is not to stop the application threads. However, the algorithm will still keep pace with the application threads for processor time. Thus, by default, the number of threads used by this GC algorithm is equal to ¼ of the number of physical cores of a particular machine[3].

The Par New GC algorithm parallelizes the copying collection over several threads, which is generally more efficient than the single-thread copying collector for multi threaded multi-processor machines. In compare to a single threaded copying collector, the algorithm accelerates `Young Generation` collection by a factor that is equal to the total available processors.

5.4. Learning Algorithm

We used several machine learning algorithms (both probabilistic and non-probabilistic) to validate our proposed approach. The algorithms include *Logistic regression (LR), Linear discriminant analysis (LDA), Naive Bayes (NB), Multilayer perceptron (MLP), and Support vector machine (SVM)*. Among all the algorithms, we mainly focus on Support vector machine due to consistent behavior with both datasets. We set the default parameters for all the learning algorithms except SVM and MLP during the experiments. In the case of MLP, we use three hidden layers and two output layers. Additionally, we set *stochastic gradient descent* as the solver and $\alpha = 1 \times 10^-5$. For SVM, we set *radial basis function* as a kernel parameter and class weight as *balanced*. Note that *Logistic regression and Naive Bayes* perform well on both original and transformed datasets.

In the following we present a brief introduction to SVM classifier which we adopted to solve our multi-class classification problem. In particular our benchmark dataset consists of five different labels due to the garbage collection algorithms. A support vector machine constructs a hyperplane or hyperplanes in a high dimensional space, which can be used for classification, regression, and others tasks. The width of separating hyperplane defines the lower generalization error margin of the SVM classifier. The standard SVM is defined for binary classification problem which can be extended for multi-class as well using "one-against-one" approach [23]. In the case of M classes, the extended binary SVM (multi-class SVM) constructs $\frac{M(M-1)}{2}$ classifiers which trains two classes each. For our particular problem, the given training data $\mathbb{X}^i \in \mathbb{R}^z$, $i = 1, 2, 3, \ldots, z$ and a vector $\boldsymbol{Y} \in \{1, 2, \ldots, 5\}^z$, SVM solves the following primal problem:

$$\begin{aligned} \min_{w,b,\zeta} \quad & \frac{1}{2} w^\top w + C \sum_{i=1}^{n} \zeta_i \\ \text{subject to} \quad & y_i(w^\top \phi(x_i) + b) \geq 1 - \zeta_i, \\ & \zeta_i \geq 0, i = 1, \ldots, z \end{aligned} \tag{5}$$

Its dual is:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \alpha^\top Q \alpha - e^\top \alpha \\ \text{subject to} \quad & y^\top \alpha = 0, \\ & 0 \leq \alpha_i \leq C, i = 1, \ldots, z \end{aligned} \tag{6}$$

where e denotes a vector of all ones $\mathbf{1}$, $C > 0$ is the upper bound, Q is a $z \times z$ positive semi-definite matrix where $Q_{ij} \equiv y_i y_j K(x_i, x_j)$. $K(.)$ is the kernel function where $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$. The training data is mapped into a higher dimensional space by the function ϕ. The decision function for classification is:

$$\text{sgn}(\sum_{i=1}^{z} y_i \alpha_i K(x_i, x) + \rho) \tag{7}$$

where ρ is the intercept parameter. In our experiment, we used linear kernel with upper bound $C = 1$.

6. Evaluation

In this section, we first describe the datasets properties with respect to different heap size. Next, we discuss the classification results and analysis for `Jolden`, `Dacapo`, and `SPEC2008` benchmarks. We use Java virtual machine version 1.8.0_161 for all experimental settings. `Dacapo` and `SPEC2008` data are being processed for the evaluations.

Figure 3 presents the garbage collection time for five garbage collection algorithms with respect to different heap settings. The settings include minimum, maximum, and initial heap allocation sizes. For example, setting 0 refers to a triple of (16,16,16) megabytes as (initial, minimum, maximum) heap size. We present the first 50 settings for visual clarity. Note that maximum heap size can not be more than the initial heap size. This rule is followed in all the experimental settings. The figure shows that Concurrent Mark Sweep algorithm performs unsteadily compared to other three algorithms. This is due to the Concurrent Mark Sweep algorithm having multiple concurrent phases. However, very few of them are not concurrent. As a result, these few non-concurrent phases are responsible to pause the currently running program. In addition, Concurrent Mark Sweep algorithm can fail if the collector does not

[3] https://plumbr.io/handbook/garbage-collection-algorithms-implementations

reclaim the garbage object before the maximum tenured generation. Moreover, there is no available space for tenured generation at the time of explicitly calling GC (e.g., `System.gc()`).

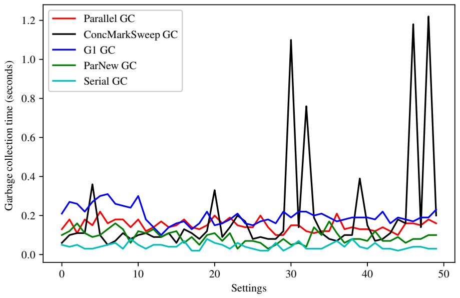


Figure 3: **Execution time behavior for "scimark" in SPEC2008 benchmark.**

Figure 4 presents the collection time for the four GC algorithms for running "compress" program in the SPEC2008 benchmark. We notice similar behavior for Concurrent mark sweep algorithm. On the other hand, Parallel GC and Serial GC algorithms perform better in terms of minimum garbage collection. However, we also notice overlap in garbage collection for different heap settings. Therefore, it is impractical to apply rule base to find the optimal GC for new test program. Hence, we use learning algorithms that adaptively select the best GC for a particular program.

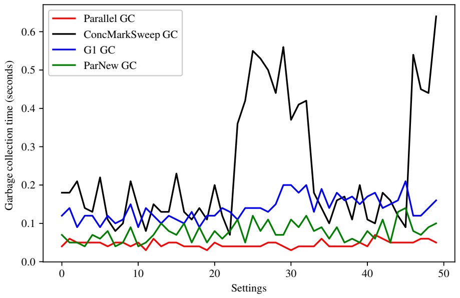


Figure 4: **Garbage collection time behavior for "compress" in SPEC2008 benchmark.**

First, we present the precision and accuracy in Figure 5 for `Jolden` data. We vary the threshold similarity between 0.5 and 0.9 that indicate the cosine similarity between any two instances (e.g., one test and another train example vector). The figure also demonstrates that both accuracy and precision have improved scores beyond similarity threshold of 0.8.

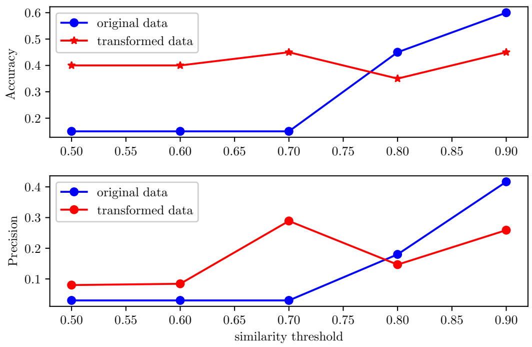


Figure 5: **Accuracy and precision score for original and transformed data with respect to varying threshold similarity (`Jolden`).**

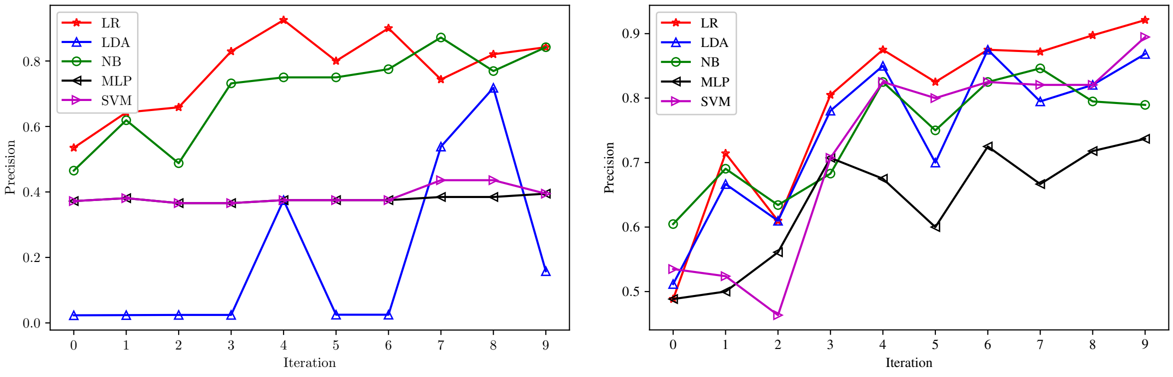
Next we discuss the precision on original `Jolden` dataset using our ***second approach*** mentioned in Section 4. Note that, we present each data using two formations such as original and transformed. The original data consists of four types of features mentioned in step 2 in Figure 1. Later, the original data is transformed using non-negative matrix factorization. The feature dimension for the transformed data is set using $d = \min(m, n)$ where dim, m, n refers to transformed feature dimension, number of rows and columns of original data, respectively.

Figure 6a presents the precision scores of different learning algorithms for `Jolden` data. From the figure, we observe that except Latent Discriminant Analysis (LDA), the other algorithms perform well with the increment in iteration. Hence, we consider LDA as baseline due to its degraded performance. We obtain Logistic Regression (LR) as a top performer in this experiment. We present the first 10 iterations for simplicity. Note that the classification is performed using 10-fold cross validation.

Figure 6b presents precision scores on the transformed data. It is evident from the result that even the worst performing classifier performs eventually better in the transformed data with the increment in iteration count. We observe that all of the considered algorithms perform well upon consideration of feature transformation. The results shows that the maximum precision of 0.90 is achieved for Logistic Regression. The main reason for this significant improvement is using a latent space that helps creating more distinguishable features.

Figure 7a presents the precision scores of the same set of learning algorithms for `Dacapo` benchmark data. Similar to the prior experiment, we also consider LDA as a baseline. The other algorithms perform remarkable with the increase in iteration count and the classification is performed using 10-fold cross validation.

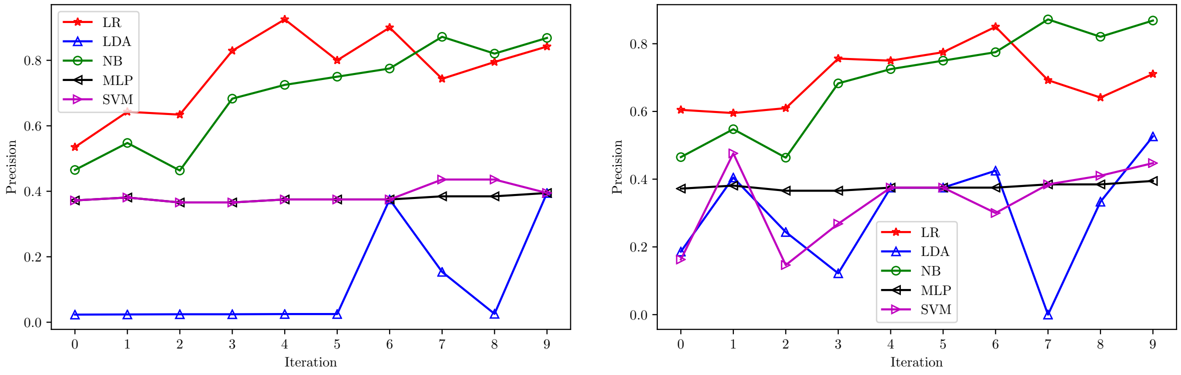
Figure 7b presents precision scores on the transformed data. The experimental result depicts that the maximum precision of 0.94 is achieved when using Naive Bayes classifier. The underlying reason



(a) **Original data.**

(b) **With feature transformation.**

Figure 6: **Precision values of different classification algorithms for `Jolden` benchmark.**



(a) **Original data.**

(b) **With feature transformation.**

Figure 7: **Precision values of different classification algorithms for `Dacapo` benchmark.**

Table 4: **Precision and accuracy scores of two approaches for two versions of learning for `Jolden` and `Dacapo` data. (max and sd values are reported)**

Benchmark	Algorithms	**Original**		**Transformed**	
		Accuracy	precision	Accuracy	precision
Jolden	ACGC+cosine (Alg1)	0.59, 0.26	0.40, 0.18	0.45, 0.20	0.28, 0.08
	ACGC+learning (Alg2)	0.84, 0.28	0.83, 0.22	0.95, 0.33	0.94, 0.32
Dacapo	ACGC+cosine (Alg1)	0.48, 0.12	0.43, 0.15	0.56, 0.18	0.52, 0.21
	ACGC+learning (Alg2)	0.86, 0.21	0.76, 0.23	0.94, 0.27	0.89, 0.28

for the improvement is similar to prior experiment that is the creation of more distinguishable features by utilizing latent space. However, we do not observe any noticeable improvement for Multi Layer Perceptron (MLP) and Support Vector Machine (SVM).

Finally, we present a simple representation of precision and accuracy scores of both approaches for `Jolden` and `Dacapo` data in Table 4. The results indicate improved performance for `ACGC` with the ML algorithms (Algorithm 2) for both benchmarks.

7. Conclusion

We posed the problem of optimal GC selection for a particular program as a supervised, binary class learning. In our formulation, the classes corresponds to existing garbage collection algorithms. We introduced a novel feature transformation method using non-negative matrix factorization. This enabled us to efficiently carry out the assignment of garbage collection algorithms to unknown programs. Our proposed methods consist of four major steps: Data collection and preprocessing, Model building, Feature generation & transformation, and Classification. One important positive aspect of our method is that the model building step is done only once before the feature generation, providing increased flexibility and computational cost reduction. We evaluated our approach using precision scores. The evaluation shows that our feature generation and classification approach outperforms baseline method. In conclusion, this paper demonstrates that `ACGC` can be posed as a feature extraction and transformation technique. Additionally, assigning unknown programs to the best garbage collection algorithm without looking at the inherent contents can be made tractable and accurate. Our proposed formulation is general and offers a potentially different mode of thinking about adaptive and concurrent garbage collection in Java virtual machine. Our implementation is publicly available in github[4]. We are planning a future work that can exploit recurrent neural network to investigate deeply to increase the prediction rate of selecting more suitable application specific GC especially to improve performance of computationally intensive programs [24, 25].

References

[1] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara, "Program-level adaptive memory management," in *Proceedings of the 5th international symposium on Memory management*. ACM, 2006, pp. 174–183.

[2] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proceedings of the 6th international symposium on Memory management*. ACM, 2007, pp. 91–102.

[3] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The garbage collection advantage: improving program locality," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 69–80, 2004.

[4] W. Hassanein, "Understanding and improving jvm gc work stealing at the data center scale," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2016, pp. 46–54.

[5] N. Cohen and E. Petrank, "Data structure aware garbage collector," in *ACM SIGPLAN Notices*, vol. 50, no. 11. ACM, 2015, pp. 28–40.

[6] E. Andreasson, F. Hoffmann, and O. Lindholm, "To collect or not to collect? machine learning for memory management." in *Java Virtual Machine Research and Technology Symposium*, 2002, pp. 27–39.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? high performance garbage collection in java with mmtk," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 137–146.

[8] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 20, 2018.

[9] S. Zobaed, M. E. Haque, S. Kaiser, and R. F. Hussain, "Nocs2: Topic-based clustering of big data text corpus in the cloud," in *2018 21st International Conference of Computer and Information Technology (ICCIT)*. IEEE, 2018, pp. 1–6.

[10] M. E. Haque, M. E. Tozal, and A. Islam, "Helpfulness prediction of online product reviews," in *Proceedings of the ACM Symposium on Document Engineering 2018*. ACM, 2018, p. 35.

[11] M. E. Haque and T. M. Alkharobi, "Adaptive hybrid model for network intrusion detection and comparison among machine learning algorithms," *International Journal of Machine Learning and Computing*, vol. 5, no. 1, p. 17, 2015.

[12] M. E. Haque, B. Al-Ramadan, and B. A. Johnson, "Rule-based land cover classification from very high-resolution satellite image with multiresolution segmentation," *Journal of Applied Remote Sensing*, vol. 10, no. 3, p. 036004, 2016.

[13] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006, pp. 169–190.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[15] W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, no. 2, pp. 155–162, 1998.

[16] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[17] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[18] A. Karczmarz, "Decrementai transitive closure and shortest paths for planar digraphs and beyond," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018, pp. 73–92.

[19] N. Bobroff, P. Westerink, and L. Fong, "Active control of memory for java virtual machines and applications." in *ICAC*, 2014, pp. 97–103.

[20] S. Soman, C. Krintz, and D. F. Bacon, "Dynamic selection of application-specific garbage collectors," in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 49–60.

[21] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "Specjvm2008 performance characterization," in *SPEC Benchmark Workshop*. Springer, 2009, pp. 17–35.

[22] S. Fenner and L. Fortnow, "Compression complexity," *arXiv preprint arXiv:1702.04779*, 2017.

[23] S. Knerr, L. Personnaz, and G. Dreyfus, "Single-layer learning revisited: a stepwise procedure for building and training a neural network," *Neurocomputing: algorithms, architectures and applications*, vol. 68, no. 41-50, p. 71, 1990.

[24] M. Haque, S. Zobaed, M. E. Tozal, V. Raghavan *et al.*, "Divergence based non-negative matrix factorization for top-n recommendations," in *Proceedings of the 52nd Hawaii International Conference on System Sciences, (HICSS'19)*, 2019.

[25] S. Zobaed and M. A. Salehi, "Big data in the cloud," in *Encyclopedia of Big Data*. Springer, 2018.

[4] `https://git.io/fjanq`