Learning When to Garbage Collect with Random Forests

Nicholas Jacek
University of Massachusetts Amherst
Amherst, MA, USA
njacek@cs.umass.edu

J. Eliot B. Moss
University of Massachusetts Amherst
Amherst, MA, USA
moss@cs.umass.edu

Abstract

Generational garbage collectors are one of the most common types of automatic memory management. We can minimize the costs they incur by carefully choosing the points in a program's execution at which they run. However, this decision is generally based on simple, crude heuristics. Instead, we train random forest classifiers to decide when to collect based on features gathered from a running program. This reduces the total cost of collection in both time and space. We demonstrate useful generalization of learned policies to unseen traces of the same program, showing this approach may be fruitful for further investigation.

CCS Concepts • **Computing methodologies → Supervised learning by classification**; **Classification and regression trees**; • **Software and its engineering → Garbage collection**.

Keywords garbage collection, machine learning

ACM Reference Format:
Nicholas Jacek and J. Eliot B. Moss. 2019. Learning When to Garbage Collect with Random Forests. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3315573.3329983

1 Introduction

Generational, compacting garbage collectors (GCs) are one of the most popular types of automatic memory management in modern language run-time systems. The cost of garbage collecting varies widely at different points in time during a program's execution, which implies that the total cost of collection can be minimized by carefully choosing the points at which to collect. Prior work [12] has shown how, given perfect knowledge of a program's entire run on a certain input, we can optimally choose these collection points. For some programs and heap sizes, the cost savings are significant.

However, this model is unrealistic for a practical system, which could only have access to a relatively limited amount of data about a program's execution so far. This raises the question: can similar benefits be realized when the decisions of when to collect are based only on this more limited information? In this work, we make strides toward answering this question in the affirmative.

First, we investigate what information could be useful to help decide when to collect. In theory, any method call or return, or any object creation, could be instrumented by a collector. However, instrumenting more than a few would impose unacceptable performance penalties. We apply *random forest importance*, a concept from machine learning, to this problem. It gives us a method of ranking how useful a certain feature is likely to be in informing us when a collection should be performed. Then, we can select a small number of features that have the highest importance.

Next, we explore how this information can be used actually to decide when to collect. We develop a proof-of-concept collection policy based on random forest classifiers. At each time step, the current features are used to decide whether, and what type, of collection to perform.

Finally, we evaluate our methods on traces of executions of Java programs. We show that for many programs and heap sizes, we can realize significant cost savings, even when our decisions are only based on a small number of features.

The main contribution of this paper is the combination of existing trace collection and optimal cost calculation techniques with machine learning methods to produce a novel algorithm for finding efficient program-specific garbage collection policies. The rest of this paper is organized as follows. The next section discussed related work, placing this work in context. In Section 3, we detail our model of a generational garbage collector and the traces we use to simulate it. Next, Section 4 contains the main contributions of this paper. It explains the machine learning techniques we employ, and how they are used to construct a collection policy. In Section 5, we give experimental results showing the effectiveness of our techniques. Finally, Section 6 outlines some directions in which this work could be extended in the future.

ISMM '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6722-6/19/06…$15.00
https://doi.org/10.1145/3315573.3329983

2 Related Work

Prior work relates to this contribution in one of three primary ways: (1) it applies machine learning to a problem in garbage collection; (2) it addresses, directly or indirectly, the problem of when to trigger a garbage collection, whether or not it uses a machine learning technique; or (3) it lays the groundwork of our approach, which requires knowledge of optimal (or perhaps near-optimal) choices of when to collect.

2.1 Machine Learning Applied to GC

Adaptive control has a venerable history in automatic memory management, so it is not surprising that researchers have used machine learning to derive adaptive GC controls of various kinds. Singer et al. [22] used random forests to select which of three GC algorithms (serial, parallel, or concurrent) and which of two young:old heap size ratios (1:2 and 1:8) to use for various inputs in the context of a specific Java application, namely Map-Reduce. Shen et al. [21] similarly used machine learning to determine which collector algorithm to use based on per-program characterization of program inputs [15]. Akram et al. [1] presented a scheme to choose adaptively which kind of core on which to run GC in a heterogeneous multi-core system. Kang et al. [14] applied reinforcement learning to a kind of GC employed within solid-state drives, reducing the occurrence of occasional long wait times for application accesses. Marion et al. [16] developed decision trees to determine whether a given allocation site should pre-tenure objects allocated at that sure, i.e., allocate the objects directly into an immortal space, a long-lived space, or a short-lived space. This list may not be exhaustive, given how extensive the GC literature is.

2.2 Triggering Collection

Most other work and deployed systems trigger GC when some allocation space is full, or when its occupancy passes some threshold. This corresponds to what we call in this work the *default* policy, namely "collect when the heap is full" (or "when the young generation is full"). In such systems, the primary way to control when to garbage collect is to control the heap size.[1] Most approaches to adjusting heap size are based on some model of adaptive feedback control: during a given program run, they observe past behavior and adjust heap size for the future. Often there are a number of user controls that can be adjusted ("knobs" one can turn), such as minimum and maximum heap size, minimum and maximum increases or decreases, or rates of increase / decrease, etc. Many deployed systems use heuristically developed adaptive heap sizing. An example is the Ergonomics system in HotSpot [23]. However, White et al. [24] found that applying control theory to the problem yielded a more responsive controller.

Previous work has addressed GC triggering, or adaptive heap sizing, for certain particular contexts, such as adaptive heap sizing to minimize battery use [6], applying temporal-difference reinforcement learning to trigger compacting or non-compacting GC in a concurrent system [2], and when to collect in an interactive system [9].

An approach that bears a family resemblance to ours is *opportunistic* collection.[2] This led to the idea that certain points in a program run will tend to be better points at which to collect, and that perhaps they could be identified, maybe even as a program runs. Hayes [8] suggested triggering collection when certain key objects become garbage, because a method holding the last reference to the object has returned. Roh et al. [20] developed a sophisticated scheme to identify and exploit different allocation phases of a program, ones where the live size is growing (ramp phase), where it is relatively constant (plateau phase), and where it suddenly decreases (cliff phase). This also involved partitioning the heap according to the phase in which an object was allocated, etc. They determined phases at run time according to key methods, which they identified by the fact that these methods' stack frames were roots for large volumes of long-lived objects.

2.3 Optimal Collection

Our prior work on after-the-fact analysis to determine which times to collect lead to optimal GC cost is essential to enabling the approach presented here. Jacek et al. [11] developed a method that determines approximately optimal choices of collection times. It used a machine learning method called least squares policy iteration, a linear method that is guaranteed to converge to an optimal model, within its range of models. However, those models do not necessarily converge to optimal *cost*. We later developed a dynamic programming approach that computes collection schedules with truly optimal cost [12]. It achieves this in $O(n^2)$ time where n is the length of a trace. Both approaches rely on a detailed trace that gives object birth times and sizes, pointer updates, and object death times, such as supplied by Elephant Tracks [18, 19]. All of this is needed in the approach taken here, so that we can train a machine learning model based on to the quality of its decisions in any given situation.

3 Model of Garbage Collection

We now describe our model of a generational collector, the nature of the traces we obtained for runs of various Java programs, how we post-processed those traces to prepare for the learning task, and offer summary statistics about the specific traces.

[1] Note that our work examines an orthogonal question: For a given *fixed* heap size, when should we trigger collection to obtain lower collection cost, or so as to require less space with the same cost?

[2] The term seems to originate from [25], who also described a collector design [26]. However, that actual scheme is more adaptive as opposed to being driven more directly by program behavior.

3.1 Model of Generational Collection

We are concerned with a *generational* garbage collector. In our model there is a young and an old generation. Each has a chosen fixed size. The old generation must be at least as large as the young one. Java objects are first allocated in the young generation. When that generation fills, or at some earlier time if the collector so chooses, the young generation may be collected, using a procedure called *young collection*. A young collection copies objects reachable from running Java threads and global variables, *and* ones reachable from objects already in the old generation, to the old generation. The old contents of the young generation are then discarded.

Another procedure the collector can apply is a *full collection*, where all objects reachable from running threads and globals are compacted together in the old generation, and the remaining contents of both generations are discarded. Notice that a young collection may preserve young generation objects that are referred to by *unreachable* old generation objects, while a full collection requires reachability from Java roots (threads and globals).

A young collection is generally cheaper than a full collection because the volume of objects examined is lower—a full collection traces through all reachable objects, and young space is smaller than old space. However, a full collection is necessary in order to reclaim space in the old generation. The collector can choose a full collection at any point, and *must* choose full collection if the space used in the young generation is more than what remains in the old generation.[3] There is also a special case: if the program desires to allocate an object that is at least as large as the young generation, then the young generation must first be emptied, by one kind of collection or the other, and the object will then be allocated directly into the old generation.

3.2 Cost Model

It is known that the running time of collectors that copy objects—the kind we consider here—is roughly proportional to the volume of objects copied, so we use the number of bytes copied as our cost model. Therefore, the cost of a young collection is the volume of objects copied from young to old space, and the cost of an old collection is the size of all the reachable objects. Other cost models are possible, such as charging a "rental" for volume of data in the heap at each time step. Our work focuses on minimizing total garbage collector effort, rather than trying to reclaim heap space as quickly as possible.

[3]This differs from the rule used by Jacek et al. [12]. That rule allows a young collection if the volume of young objects that will be copied will fit in old space. However, this fact cannot always be known until after one attempts the young collection. While it is possible to imagine collectors that can recover from the situation of copies that will overflow old space (see, e.g., McGachey and Hosking [17]; it is not clear how widely this is used), it seems more typical to apply the more conservative policy we use here, because it simplifies collector implementation. The adjustment to the dynamic program of Jacek et al. [12] was straightforward and did not impact the asymptotic running time of that algorithm, etc.

3.3 Collection Schedules and Polices

We next consider our GC model from a Reinforcement Learning point of view. At each time step in the execution of a trace, we want to select the action—no collection, young collection, or full collection—that yields the lowest cost over the entire trace. The algorithm of Jacek et al. [12] computes the optimal schedule: a list of which action is optimal at each step. However, this is only useful for a single program and input. A list of optimal actions for one trace gives us no information about how to choose good actions in a different trace. Instead, we seek a collection policy—a function that takes measurable features of a program's execution as input and gives predictions of the optimal action as output. A good policy would use these features to give action predictions for previously unseen inputs to a program.

This is a difficult task. By collecting early, we can select points that have lower volumes of live data and thus lower costs. However, early collections can lead to more frequent collections, and the cost of these additional collections can eliminate any savings. Furthermore, the exact points at which collections are performed influence which objects are promoted in future young collections. These effects are complicated and may not be realized until many steps in the future. In past unpublished studies we found that *locally* optimal collection, i.e., choosing a point to collect between the current time and the time when young space has just been filled, such that young space collection cost is minimized, does *not* lead to *globally* optimal collection. Globally optimal collection is a combinatorial optimization problem, and it requires exact knowledge of future behavior. However, given features about the execution of a program, such as we use in the work reported here, gives hope of reasonably accurate prediction of that future behavior, at least for some programs.

3.4 Nature of Our Traces

We used the Elephant Tracks (ET) tool [18, 19] to obtain sequences of event records from executions of Java programs. There are two kinds of relevant events in these traces: control events (method calls, returns, etc.) and heap events (object allocation, pointer updates, object death). Like its intellectual predecessor Merlin [10], ET computes precise death times for each object, i.e., the point at which the object became no longer reachable. ET's strategy for this is to record when references to heap objects are destroyed. The point of destruction is a possible end of reachability for the referent; in any case, the referent was definitely *reachable* up until the point of destruction. ET then determines death time as the last time an object was reachable from a root.

We post-process these event traces in two significant ways:

1. By simulating the heap events (allocations, pointer updates, and deaths), we can determine the pointers within each object when it dies. We then use that information to compute the *pre-birth* time for each object, a concept introduced by Jacek et al. [12]. Consider conceptually a heap large enough to hold all objects allocated during a program's execution. Suppose we inspect that heap at the end of the run and determine, for a given object o, the set of objects from which we can reach o in the heap, i.e., the predecessors of o in the heap graph. (The heap graph is the directed graph where objects are nodes and pointers are the directed edges.) The pre-birth time of o is the minimum (earliest) of the birth (allocation) times of o's predecessors.

   Knowing the pre-birth time is significant because it enables direct determination of whether a given young collection will preserve o. Of course o will be preserved if it is *live* (reachable) at the time of the collection—a requirement of garbage collector correctness. However, if the previous collection (young or full) occurred between the pre-birth and birth time of o, and the next collection is a young collection after o dies, o will also be preserved. (In the terminology of Jacek et al. [12], this is the cases in which o is *baggage*.)
2. We *group* events of the trace. Every 256 Kbytes of allocation forms an allocation group, and we also group the control events that occur in the same interval. Specifically, if adding a second or later object to the current group would cause the group's size (bytes allocated) to exceed 256 Kbytes, then the allocation event of that next object starts a new group. Groups are reasonable in that most real allocators will make a decision about whether to run the garbage collector only as a block of some size fills. Groups serve several purposes in this work.
   - Groups facilitate calculating optimal collection schedules efficiently, as discussed by Jacek et al. [12], because we can deal with whole sets of objects at once rather than handling each object separately.
   - Pre-computed groups further insure consistent definition of the possible times for collection across heap sizes and previous collection histories.
   - Groups are our basis for defining feature vectors from the control events of a trace. In particular, within a group we compute for each Java method two features concerning calls of that method: the number of times the method was called, and a 0/1 feature that indicates just whether the methods was called at all. Each source of control events determines a similar pair of features: calls of a method, returns from the method, exception throws and catches, calls from a given call site, and allocations at a given allocation site. We also have features for the number of bytes allocated at an allocation site, and the number of array elements allocated at a site that allocates arrays.

     A given program may have tens of thousands of possible features. Typically only about 10% of the possible features are actually used in a given execution, and of course many of those are zero in the time window of a given group. Still, the number of features is large, so ultimately it is important to control how many are used in a learned policy function. This is true both because obtaining a feature's value at run time has a cost every time the feature's event occurs, and because evaluating the learned function will be costly if it uses a large number of features.

3.5 Trace Products

There are two key post-processed products for each trace:

Allocation cohorts: A *cohort* is a set of objects whose pre-birth times fall into the same group, whose birth times fall into the same group, and whose death times fall into the same group. Because those times entirely determine the collector's behavior and costs (under our models), for modeling collection behavior all we need to know is the set of cohorts and their sizes.

Feature vectors: Concerning the feature vectors, feature numbers for the same method may vary from trace to trace, since the overall set of methods can be different. When we handle multiple traces from the same program, we first map all the features of the individual traces onto the union of features across the traces.

An additional trace product is the number of times instrumentation would be triggered for each feature, i.e., an estimate of the relative cost to obtain that feature at run time. (At present we do not exploit this information.)

3.6 Trace Details

Table 1 lists the programs from which we gathered traces, indicating the number of traces for each program and the ranges of number of groups, number of cohorts, bytes allocated, and maximum live size (maximum number of bytes reachable at once) for the traces of that program. Except for javac, these are all from the DaCapo benchmark suite [4], though we developed additional inputs for most of them. In the case of javac, the program is a modified version of the original SPECjvm benchmark of the same name, but modified to avoid caching of class file information across compilation of multiple classes, to simulate better what a compilation server might be like. Across the programs there is considerable variation in the statistics, and for many of the programs considerable variation across traces.

Table 1. Summary of traces used

Program	Traces	Groups	Cohorts	Alloc (MB)	Max Live (MB)
avrora	4	181– 1606	944– 19603	45– 400	3.6– 105.7
batik	11	99– 586	586– 4766	25– 186	5.0– 24.6
fop	12	124– 6560	1024– 47124	30– 1587	5.0– 38.3
javac	4	1860–15759	25410–308523	459– 3899	10.3– 16.5
luindex	5	26– 27	170– 183	6– 6	2.0– 2.0
lusearch	3	2744–43921	9936–155178	664– 10510	2.0– 2.2
pmd	19	53– 5043	358–134539	13– 1202	1.7– 168.0

4 Learning Algorithm

Our algorithm reduces the problem of finding an efficient policy to a supervised classification problem. We treat each action as a class label. For a given program and input, the dynamic programming algorithm of Jacek et al. [12] provides us with a method of calculated the optimal action, and thus ground-truth label, of each point in the trace. We can use this data to train any supervised classification model. Afterwards, the model can be used to classify points from traces of new inputs to the program. The predicted class of each time step gives us a prediction of what the optimal action is at that step. The classifier therefore serves as our collection policy.

However, the total number of features in our traces is far greater than what could be used in a practical system. Our features are counts of method calls and returns, and object allocations. In a production system, the compiler would have to insert instrumentation that increments a counter at each point that corresponds to one of the features we use. Clearly, if we use too many features in our policy the cost of this instrumentation will quickly outweigh the benefit of our improved collection policies.

To overcome this, we first perform *feature selection* via random forest importance to decide which features to use. In this method, a large number of decision trees are constructed using random subsets of our training data and the full set of possible features. The importance of each feature is then a measure of to what extent the accuracy of these trees depends on each feature. We then select a small set of the most important features.

We also investigated different feature selection methods based on the regularization of linear models. Initial experiments suggested, however, that the linear models were not powerful enough to capture useful policies accurately. Principle Component Analysis, another popular method of feature extraction, is not appropriate for our uses. It calculates a new, smaller set of features from linear combinations of the existing features. In the problem we investigate, in order to construct these new features, we would still have to instrument all of the original features.

After selecting the most important features, we use our reduced sets of features to build our actual classifiers with random forests [5]. This is somewhat similar to our feature selection step. But, when we construct the decision trees that make up our collection policies, we restrict them to using only the reduced set of selected features.

4.1 Hellinger Trees

Our learning algorithm is based on decision trees. Each internal node of the tree holds a feature ID number and a threshold. To decide which action to take at a time step, we begin at the root node, and compare the current value of the feature against the threshold. If it is less than or equal, we proceed to the left child of the node, and if not, we proceed to the right child. Eventually we reach a leaf, which lists the probability that each action should be taken.

Decision trees are built recursively by greedily selecting the feature and threshold that best splits the examples to be classified into left and right groups according to some measure. The most common measures, Gini impurity and information gain, are not suitable for our purposes. This is because the distribution of the actions in our data are highly skewed. In optimal schedules, most of the time no collection should be performed. Young collections are rare, and full collections rarer still.

To overcome this limitation, we rank splits according to the Hellinger distance, which is much more robust to skewed distributions [7]. To see how it is calculated, consider the case where we are distinguishing between only two classes; denote them − and +. A given split creates two conditional probability distributions. Each represents the chance that an example will be passed along to either the left or right child, given that it belongs to a given class. The PMF (probability mass function) of one distribution is given by $P(L|-)$ and $P(R|-)$, and the PMF of the other is given by $P(L|+)$ and $P(R|+)$. The Hellinger distance is then one measure of the distance between these two distributions. The calculation is:[4]

$$d_H = \sqrt{1 - \sqrt{P(L|-)P(L|+)} - \sqrt{P(R|-)P(R|+)}}. \quad (1)$$

To visualize the Hellinger distance, note that these two distributions form vectors in a two-dimensional space. The

[4]This formula differs by a constant factor from what is often seen in the literature. Because we are interested only in finding the point that maximizes the Hellinger distance, the constant factor is irrelevant for our purposes.

Hellinger distance is then the Euclidean distance between the square roots of these two vectors. It reaches its maximum value, 1, when the two distributions are completely disjoint, and is zero if the distributions are equal. We select the split with the largest distance between the distributions it induces. Note, however, that we have three actions, and therefore three classes. This gives us three different conditional probability distributions. There is little guidance in the literature as to the best way to combine the distances between these three points into a single measure of the quality of a split. We take what is arguably the simplest option, and use the sum of the three pairwise distances between the three distributions.

As we build the tree we split the examples into smaller and smaller groups. Eventually, either a given group contains only one action, or no features and thresholds will further subdivide it. In this case, we add a leaf that records the distribution of actions in the group. We use Laplace smoothing, a technique that helps to avoid overfitting. It adds one to the count of examples in each class. This reduces the confidence the classifier has in the empirical probabilities when the number of examples at a leaf is small, and ensures that at every leaf every action has at least a small probability.

4.2 Importance

Importance is a concept originally developed to investigate the internal workings of random forests [5], but that has also been used as a method of feature selection [3]. For each tree of a random forest, a training set is chosen by randomly selecting examples from the full training data with replacement. The examples that are not selected then naturally form a different random test set for each tree. We can use these sets to calculate the importance of each feature.

First, we calculate the classification accuracy of the tree on the test examples. Then, we randomly permute the values that a certain feature takes among these examples, and we calculate the tree's classification accuracy again. When these two are compared, the more that permuting the values decreases the classification accuracy, the more important we conclude the feature is to the classifier. The overall importance of a feature is then the average of its importance to each tree in the random forest. We then select the most important features for use later in our final classification algorithm.

Note that it is typical to build each tree in a random forest using a random subset of features, as well as a random subset of training examples. When we attempted to do this on our data, most trees did no better than random at classifying training examples. So, we were unable to calculate meaningful importance values. This is likely because, as discussed in Section 5, the vast majority of the features do not carry information relevant to our task. Instead, as we calculate feature importance, each node in a tree is able to select greedily a feature and threshold for its split from the entire set of features and values in the data.

4.3 Learning a Policy with Random Forests

Once we have calculated importance values for all the features in the data, we select a small set of the most important features, and use them to learn our collection policies. These also take the form of random forests of Hellinger trees, similar to what were used to calculate feature importance. However, the feature used at each node of the trees is selected uniformly at random, though the threshold is still selected to maximize the Hellinger distance of the resulting split. It is well known that the performance of ensemble classification algorithms is proportional to the strength of the individual classifiers, but inversely proportional to the correlation between them. Using our reduced set of features, the classification trees have reasonable strength even when the feature at each node is chosen randomly. This random selection also decreases the correlation between the individual trees, improving the overall classifier.

Once the random forest is built, it is straightforward to use it as a policy. At each time step, we run the current features of the trace though all the trees in the forest, each of which yields a probability that each action should be taken. Then, we take the average of these probabilities and select the most probable action. In case the predicted action is not feasible according to our GC model, we simply fall back to the default policy.

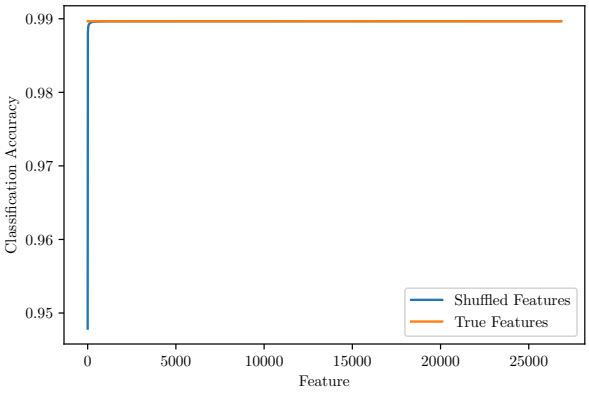
5 Empirical Results

5.1 Importance

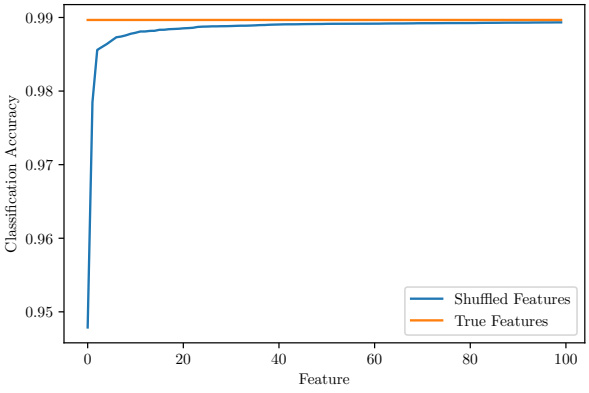
One importance calculation is performed for each program; the traces for all the inputs to the program are combined into one large training set. For each program, we build a forest of 100 trees. Each tree is trained on a random set of examples generated by sampling with replacement from the total training set and equal to it in size.

Figure 1 shows the importance values for the javac program, though the results for every program in our suite are very similar. Clearly, the overwhelming majority of features are not useful for building a policy. Most of the information about which action should be taken is contained in the several dozen top features. The high classification accuracy, even using randomly shuffled feature values, is likely due to the highly skewed nature of our data. Simply predicting that no collection should be performed at any time step is correct a large amount of the time.

In addition to the features detailed in Section 3, we add two: the fractions of the young space and old space that are full. These turn out to have particular noteworthiness. For every program in our suite but one, the fraction of the young space used is the most important feature, and in the remaining case it is 3rd most important. The fraction of the



(a)



(b)

Figure 1. Importance values for the javac program. Note the truncated vertical axis.

old space used ranges from 3rd to 13th most important. We conclude that any practical GC policy would need to use these features. In fact, the default policy is based *only* on these features.

To give a sense of the features chosen according to our importance method, here is a list of the ten most important features for our javac benchmark:

Kind	Feature
fraction	young space used
# calls	javac/util/DefaultFileManager.flush
# calls	javac/util/Log.hasDiagnosticListener
fraction	old space used
any returns	javac/util/Name$Table.dispose
# calls	javac/util/List.nil
# elements	javac/util/Convert.utf2string12e2:[C
any calls	java/lang/Double.doubleValue
# returns	javac/main/JavaCompiler.close
# calls	javac/util/List$3.hasNext

Here, javac refers to the com.sun.tools package, # refers to a counted feature, *any* refers to a 0/1 feature, and *elements* means the number of elements allocated in arrays of the given type at the given call site. Some of these features, such as JavaCompile.close, suggest significant phase changes, such as being done with compiling one class file and moving on to the next one, while others would require deeper analysis and profiling to tease out why they are important for this learning task. In any case, this illustrates the usual virtue of decision trees: they tend to lead to models more interpretable by humans than do many other learning methods.

The time to perform the importance calculation is quite large due to the extreme number of features that must be considered at each split in each decision tree. We list the times taken for our calculations in Table 2.

5.2 Policy Performance

For policy training, we selected the 32 instrumented features with the highest importance as an approximation of the number that could be used in a practical system. The fraction of the young and old spaces that are occupied would also be available to a system so they are also included, bringing us to a total of 34 features. First, we trained a single decision tree on each trace in our set, and then tested its performance on that same trace. In all cases, the learned trees were able to perfectly reproduce the optimal policy. This likely indicates that they were overfitting to their training data, but it does suggest that the selected features are informative enough to represent good policies.

Next, we investigated how well our learned policies could generalize across different inputs to a program. For this we used random forests, and each was built with 1000 trees. We selected this large number in order to give our policies the best chance of performing well, since random forests do not tend to overfit as the number of trees they contain is increased. A practical system would require additional engineering to reduce this number and minimize the time the policy needs to make its decision. Still, our results give insight into what might be possible for any practical system.

We performed leave-one-out cross validation of our policy by training a random forest on all but one input to a program. The resulting policy was then simulated on the remaining input. We repeated this procedure for every input in our suite.

Our training data represented optimal schedules for old spaces sized to be three times the maximum live size of the trace. For all programs except luindex, we used a young space size of 8MB. This size is large enough that luindex did not perform any collections, so instead we used a young space size of 2MB for that program only.

We tested each of our learned policies on 100 different old sizes ranging from 1 to 5 times the maximum live size of the

Table 2. Time to perform importance calculations

Program	avrora	batik	fop	javac	luindex	lusearch	pmd
Time (hrs)	0.48	2.24	18.93	27.75	0.010	20.09	31.36

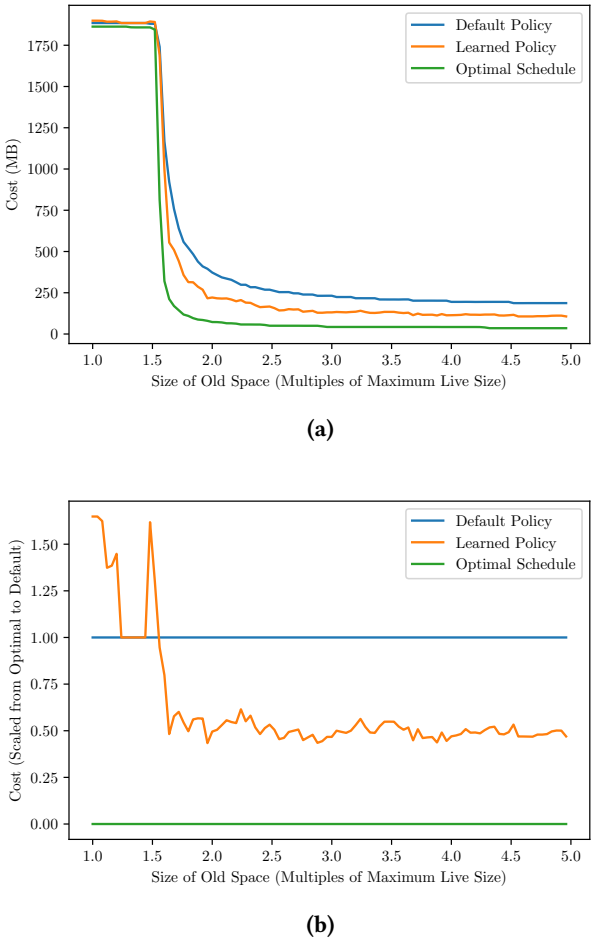


Figure 2. Policy costs for javac asm.

trace. The results from one program and input, javac on asm, are given in Figure 2.

Figure 2(b) shows the same data as Figure 2(a), but the cost values have been shifted and scaled to bring the cost of the optimal schedule to 0 and the cost of the default policy to 1. The graphs show that the performance of the learned policy does not generalize well to small heap sizes, but for most of the range it achieves about half the possible improvement.

Unfortunately, not all traces fare as well. Figure 3 shows similar graphs for the large60 input to the batik program. For most of the range, the learned policy simply reproduces the default policy. For some traces, it does so for its entire range. However, the performance of the learned policies varies between inputs to a single program, not simply across different programs. Figure 4 illustrates a different input to the batik program on which the learned policy has much better performance than the default.

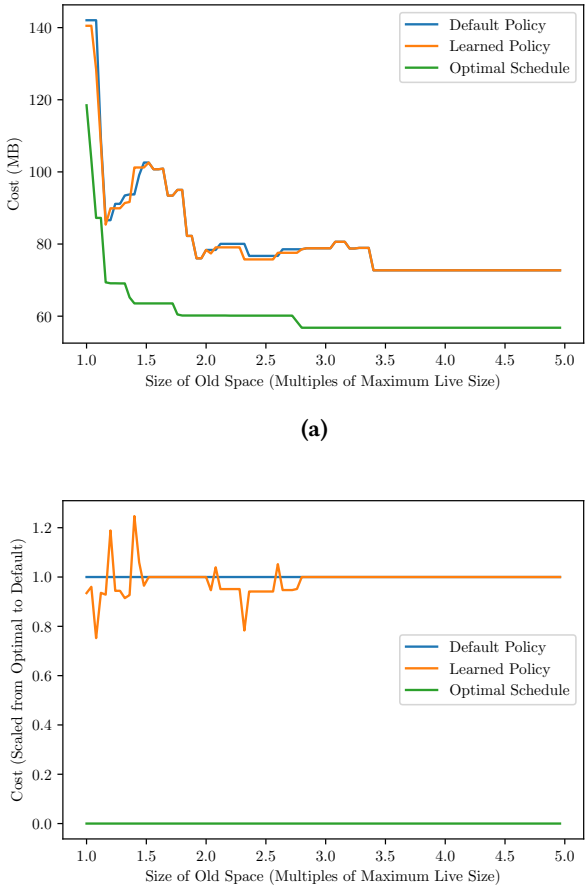
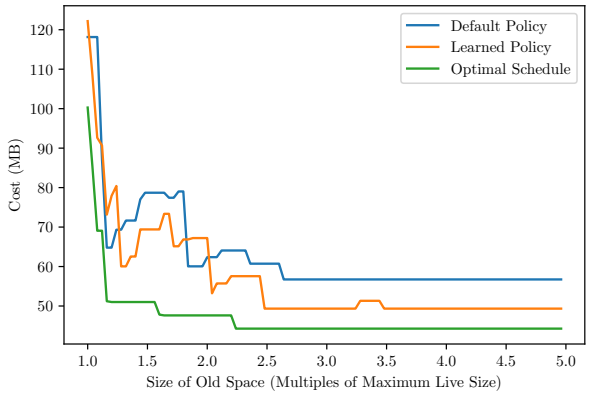


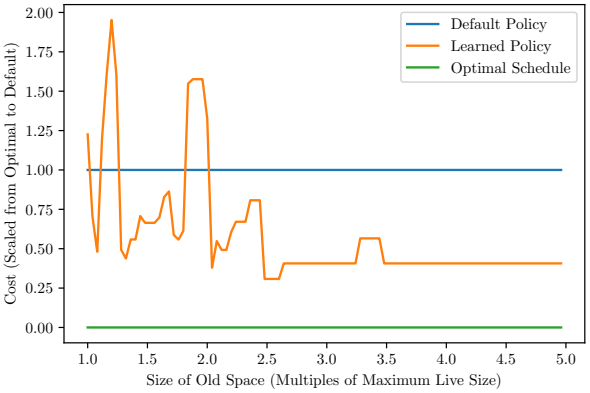
Figure 3. Policy costs for batik large60.

Next, Figure 5 gives the distribution of scaled costs for every program and input in our suite. About half the time, the learned policy is identical to the default policy, and in rare cases worse. However, in many cases the learned policy is better. It gives a mean of about 20 percent of the possible improvement so long as the heap size is not too small. In a few cases, it equals the optimal schedule.

Then, Figure 6 presents the same data with the costs scaled to multiples of the optimal cost with a heap size of three times the maximum live size of the trace, the size at which the policy is trained. It shows that the costs of all three policies



(a)



(b)

Figure 4. Policy costs for batik default60.

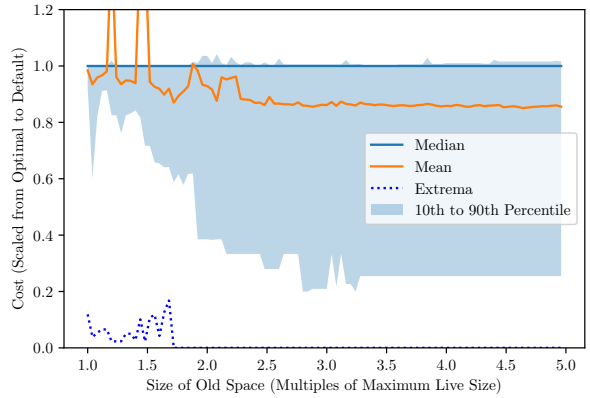


Figure 5. Distribution of learned policy costs.

are greatest with small heap sizes. As the heap size increases, the costs of the policies slowly approach each other, with

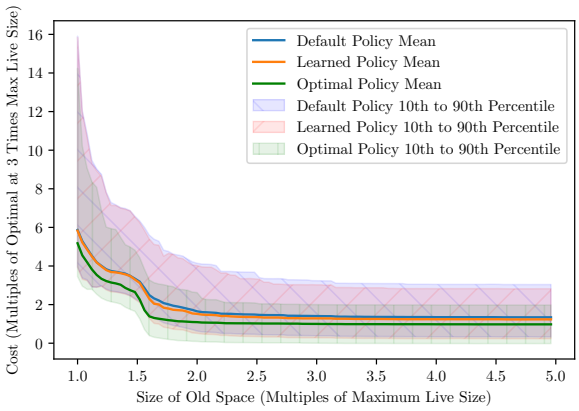


Figure 6. Distribution of learned policy costs.

the cost of the learned policy remaining between those of the default policy and optimal schedule.

Finally, we show the time needed to train our random forest classifiers in Table 3. The greatly reduced set of features leads to much faster times compared to the importance calculation.

6 Conclusions

We have shown that it is possible to learn GC policies that improve collector performance even on new inputs. To our knowledge, this is the first work in the literature to apply machine learning techniques to the problem of optimizing generational garbage collection times.

A number of avenues remain open for future research:

- There are different performance costs to instrumenting different features. A practical policy may benefit from selecting features in a cost-aware manner.
- We have simply selected the most important features, but some of these may be highly correlated with each other and thus redundant. A more sophisticated selection scheme may result in a more informative set of features, and give better results for the same *number* of features.
- Classifiers other than random forests may give better performance on this task. For example, our random forests classify each time step on its own and do not make any use of the fact that our data form ordered series. Likewise, it is possible that neural net models, for example, might do better (although a quick check suggested that in this simple case they did worse).
- For each program and input, our classifiers are trained on a single trace using a single heap size. Including additional heap sizes in the training data may help the learned policies to generalize to new inputs and heap sizes. This may address the issue of the sometimes bad performance for small heap sizes.

Table 3. Time to train random forest classifiers

Program	avrora	batik	fop	javac	luindex	lusearch	pmd
Minimum Time (mins)	0.25	0.90	2.99	4.57	0.022	3.01	11.41
Median Time (mins)	0.59	1.04	5.28	11.01	0.024	9.60	12.81
Maximum Time (mins)	0.69	1.22	6.54	14.97	0.027	9.98	15.44

- The policies are also given only the exactly optimal schedules as ground truth, but many schedules have costs only slightly higher. Training on these slightly sub-optimal schedules as well may improve the performance of our policies by offering more training data and avoiding over-sensitivity to the exact decisions needed to achieve optimal cost.

Much work remains before collection policies similar to those we investigate here could be put into production in real runtime systems. We have, however, made the important first steps toward using machine learning techniques to decide when generational garbage collectors should be run in order to optimize their performance, and we have shown that further investigation may be worthwhile.

Acknowledgments

This work is supported by the National Science Foundation under grant CCF-1320498. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. We thank an anonymous reviewer, and our colleague Richard Jones, for suggesting related work to mention.

References

[1] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2016. Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors. *ACM Transactions on Architecture and Code Optimization* 13, 1 (March 2016), 4:1–4:25. https://doi.org/10.1145/2875424

[2] Eva Andreasson, Frank Hoffmann, and Olof Lindholm. 2002. To Collect or Not to Collect? Machine Learning for Memory Management, See [13]. http://www.usenix.org/events/jvm02/andreasson.html

[3] Kellie J Archer and Ryan V Kimes. 2008. Empirical characterization of random forest variable importance measures. *Computational Statistics and Data Analysis* 52, 4 (2008), 2249–2260.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006*, Peri L. Tarr and William R. Cook (Eds.). ACM, Portland, OR, 169–190. https://doi.org/10.1145/1167473.1167488

[5] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[6] Gungyu Chen, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Mario Wolczko. 2002. Adaptive Garbage Collection for Battery-Operated Environments, See [13], 1–12. http://static.usenix.org/event/jvm02/full_papers/chen_g/chen_g.ps

[7] David A Cieslak, T Ryan Hoens, Nitesh V Chawla, and W Philip Kegelmeyer. 2012. Hellinger distance decision trees are robust and skew-insensitive. *Data Mining and Knowledge Discovery* 24, 1 (2012), 136–158.

[8] Barry Hayes. 1991. Using Key Object Opportunism to Collect Old Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (ACM SIGPLAN Notices 26(11))*. ACM Press, Phoenix, AZ, 33–46. https://doi.org/10.1145/117954.117957

[9] Roger Henriksson. 1996. *Adaptive Scheduling of Incremental Copying Garbage Collection for Interactive Applications*. Technical Report 96–174. Lund University, Sweden. ftp://mjolner.dna.lth.se/HD/ftp/pub/papers/LU-CS-TR.96-174.ps

[10] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. 2006. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems* 28, 3 (2006), 476–516. https://doi.org/10.1145/1133651.1133654

[11] Nicholas Jacek, Meng-Chieh Chiu, Benjamin M. Marlin, and Eliot Moss. 2016. Assessing the limits of program-specific garbage collection performance. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, Santa Barbara, CA, 584–598. https://doi.org/10.1145/2908080.2908120

[12] Nicholas Jacek, Meng-Chieh Chiu, Benjamin M. Marlin, and J. Eliot B. Moss. 2019. Optimal Choice of When to Garbage Collect. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 3 (Jan. 2019), 35 pages. http://doi.acm.org/10.1145/3282438

[13] JVM 2002 2002. *2nd Java Virtual Machine Research and Technology Symposium*. USENIX, San Francisco, CA. http://www.usenix.org/event/jvm02

[14] Won-Kyung Kang, Dongkun Shin, and Sungjoo Yoo. 2017. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Transactions on Embedded Computer Systems* 16, 5 (2017), 134:1–134:20. https://doi.org/10.1145/3126537

[15] Feng Mao and Xipeng Shen. 2009. Cross-Input Learning and Discriminative Prediction in Evolvable Virtual Machines. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization*. IEEE Computer Society, Seattle, WA, USA, 92–101. https://doi.org/10.1109/CGO.2009.10

[16] Sebastien Marion, Richard Jones, and Chris Ryder. 2007. Decrypting the Java Gene Pool: Predicting Objects' Lifetimes with Micro-Patterns. In *6th ACM SIGPLAN International Symposium on Memory Management*, Greg Morrisett and Mooly Sagiv (Eds.). ACM Press, Montréal, Canada, 67–78. https://doi.org/10.1145/1296907.1296918

[17] Phil McGachey and Antony L. Hosking. 2006. Reducing generational copy reserve overhead with fallback compaction. In *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006*, Erez Petrank and J. Eliot B. Moss (Eds.). ACM, Ottawa, Ontario, Canada, 17–28. https://doi.org/10.1145/1133956.1133960

[18] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. Tool Demonstration: Elephant Tracks—Generating Program Traces with

Object Death Records. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*. ACM, ACM, Kongens Lyngby, Denmark, 39–43.

[19] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable production of complete and precise GC traces. In *International Symposium on Memory Management, ISMM '13, June 20, 2013*, Perry Cheng and Erez Petrank (Eds.). ACM, Seattle, WA, 109–118. https://doi.org/10.1145/2464157.2466484

[20] Yangwoo Roh, Jaesub Kim, and Kyu Ho Park. 2009. A Phase-Adaptive Garbage Collector Using Dynamic Heap Partitioning and Opportunistic Collection. *IEICE Transactions on Information and Systems* E92-D, 10 (Oct. 2009), 2053–2063.

[21] Xipeng Shen, Feng Mao, Kai Tian, and Eddy Zheng Zhang. 2009. The Study and Handling of Program Inputs in the Selection of Garbage Collectors. *SIGOPS Operating Systems Review* 43, 3 (July 2009), 48–61. https://doi.org/10.1145/1618525.1618531

[22] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. 2011. Garbage Collection Auto-Tuning for Java MapReduce on Multi-cores. In *10th ACM SIGPLAN International Symposium on Memory Management*, Hans Boehm and David Bacon (Eds.). ACM Press, San Jose, CA, 109–118. https://doi.org/10.1145/1993478.1993495

[23] David Vengerov. 2009. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *8th ACM SIGPLAN International Symposium on Memory Management*, Hillel Kolodner and Guy Steele (Eds.). ACM Press, Dublin, Ireland, 1–9. https://doi.org/10.1145/1542431.1542433

[24] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. 2013. Control Theory for Principled Heap Sizing. In *12th ACM SIGPLAN International Symposium on Memory Management*, Erez Petrank and Perry Cheng (Eds.). ACM Press, Seattle, WA, 27–38. https://doi.org/10.1145/2464157.2466481

[25] Paul R. Wilson. 1988. Opportunistic Garbage Collection. *ACM SIGPLAN Notices* 23, 12 (Dec. 1988), 98–102.

[26] Paul R. Wilson and Thomas G. Moher. 1989. Design of the Opportunistic Garbage Collector. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (ACM SIGPLAN Notices 24(10))*. ACM Press, New Orleans, LA, 23–35. https://doi.org/10.1145/74877.74882