On The Limits of Modeling Generational Garbage Collector Performance

Peter Libič* Lubomír Bulej† Vojtěch Horký* Petr Tůma*

*Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University, Czech Republic
{libic,horky,tuma}@d3s.mff.cuni.cz

†Faculty of Informatics, University of Lugano, Switzerland
lubomir.bulej@usi.ch

ABSTRACT

Garbage collection is an element of many contemporary software platforms whose performance is determined by complex interactions and is therefore difficult to quantify and model. We investigate the difference between the behavior of a real garbage collector implementation and a simplified model on a selection of workloads, focusing on the accuracy achievable with particular input information (sizes, references, lifetimes). Our work highlights the limits of performance modeling of garbage collection and points out issues of existing evaluation tools that may lead to incorrect experimental conclusions.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements; D.4.2 [**Storage Management**]: Garbage collection

General Terms

Performance

Keywords

performance modeling; garbage collector; java

1. INTRODUCTION

A garbage collector (GC) is an essential part of modern runtime platforms. Whether used in mature virtual machine environments such as Oracle Java Virtual Machine (JVM), and Microsoft Common Language Infrastructure (CLI), in functional languages such as Lisp, and Haskell [15], or in dynamic languages such as Ruby, and JavaScript, the GC plays a major role in the overall system performance.

There are many ways to implement a GC—the design space comprises different algorithms and parameter configurations [10], but there is no single GC that works best for all applications. Some garbage-collected environments provide multiple GC implementations—either to enable workload-specific tuning (Oracle HotSpot JVM), or to facilitate experimental variability for development and evaluation of GC algorithms (Jikes RVM).

The choice of a GC and its configuration can have significant impact on application performance [6], but due to complex interactions between the GC and the application workload, there are no exact guidelines (or algorithm) telling a developer what GC to choose and how to configure it. Instead, developers are given recommendations for trial-and-error tuning [11, 17, 16].

While many developers embraced the GC-based platforms due to increased productivity brought about by automatic memory management, the influence of a GC on application performance is significantly less well understood. Indeed, even in the performance engineering community, the GC overhead is often modeled as a constant factor to be calibrated with other model parameters [5, 25]. However, the accuracy of such models is at best difficult to ascertain, because anomalous GC overhead under certain workloads can account for tens of percents of execution time [12].

Assuming the perspective of an application developer with knowledge of GC principles—but very limited influence on particular GC internals—our goal is to determine whether the developer can get a reasonably intuitive understanding of GC performance, which would allow to relate GC behavior to application-level performance and vice versa. Mismatches between the observed and expected application-level performance would indicate situations where special attention is needed, especially if predictable performance is desired.

To this end, we investigate the performance behavior of a real GC implementation compared to a simplified model implemented as a GC simulator. In particular, we evaluate the model accuracy on a variety of workloads and perform sensitivity analysis with respect to the input describing the application workload. Our contribution is as follows:

– We define simplified models of a one-generation and a two-generation GC, and evaluate their GC prediction accuracy on a variety of workloads, showing surprisingly good results for some of them.

– We analyze how the prediction accuracy depends on the information present in the input data, and discuss the

ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2568097.

results in light of the complex interactions that govern the behavior of contemporary garbage collectors.

– We highlight the limits of GC performance modeling, pointing out issues that hinder experimental evaluation and that may lead to incorrect conclusions with existing tools.

The paper is structured as follows: we complement the introduction with a short overview of related work in Sect. 2. In Sect. 3, we present the general approach applied to modeling one-generation GC in Sect. 4 and two-generation GC in Sect. 5. We analyze model sensitivity to reduced and inaccurate input in Sections 6 and 7, respectively, and conclude the paper in Sect. 8.

2. RELATED WORK

The primary source of information on GC performance is the GC research community—a GC design necessarily needs to make many low-level design decisions related e.g. to barriers, data structures, or traversal algorithms [9, 2, 22, 4, 7], and the overhead of every proposed GC algorithm is carefully evaluated. Despite the insights provided by the research on GC design, it is difficult for both developers and performance engineers to relate the knowledge of GC internals to application-level performance. To gain information on observable performance, benchmarks from established benchmark suites such as SPECjbb, SPECjvm, or DaCapo are typically used. Of these three, the SPECjbb and SPECjvm suites are intended for general JVM performance evaluation, while the DaCapo suite is designed to exercise the GC [6].

A significantly smaller body of work can be found in the area of GC modeling. Since object lifetimes play a major role in GC behavior, an efficient algorithm for collecting the lifetime data has been developed [13] and implemented [20]—yet attempts at modeling garbage collector performance are rare. The performance engineering community typically models GC overhead as a constant factor [5, 25], while more specific models can be mostly found in the domain of GC parameter optimization. Vengerov [23] derived an analytical model for the throughput of the generational GC in the HotSpot JVM, which allows optimizing the sizes reserved for the young and old object generations. White et al. [24] used a control-theoretic approach (a PID controller) to adapt the heap size in response to measured GC overhead.

3. GENERAL APPROACH

In general, our approach is based on comparing the behavior of a GC model to the behavior of a real GC implementation. We consider both a simple one-generation GC and a more common two-generation GC. For each GC type, we define a simplified model based on the principles inherent to that particular type. Compared to a real GC implementation, the model omits technical details (such as what the barriers look like or how the GC manages used and free memory) that an application developer would be unlikely to care about or unable to control.

We use the frequency of garbage collection cycles as the metric to evaluate the model accuracy on. We investigate the reasons for mismatches between the modeled and observed behavior—from the application developer perspective, these mismatches indicate situations where the GC behavior cannot be explained based on the intuitive understanding of the basic principles of GC operation. Knowing what the underlying cause for the mismatch is allows the developer to either look for a GC that behaves more predictably, or adapt the application code to avoid triggering the behavior.

To analyze the sensitivity of the model to the input describing the application workload, we compare the behavior of the real and simulated GC with different inputs, ranging from complete traces (containing object lifetimes, object sizes, and reference updates) to minimal input in form of probability tables (capturing object lifetime and size distributions). In contrast to the existing work, we measure object lifetime in total object allocations, instead of method invocations [20], or total bytes allocated [13].

At this stage, we do not attempt to model GC overhead in terms of execution time, because that is virtually impossible without getting the fundamental metrics right and thus being able to tell when a collection occurs. Our initial experiments suggest that the duration of individual collections is often in an approximately linear relationship with the number of objects surviving the collection, but we defer detailed investigation to future work.

4. ONE-GENERATION COLLECTOR

To validate the feasibility of our approach, we first consider a one-generation GC and build a simplified model with the following assumptions: (a) objects have headers and observe address alignment rules, (b) objects are allocated sequentially in a single heap space, (c) garbage collection is triggered when the heap runs out of free space, (d) all unreachable objects on the heap are reclaimed in a single GC run, and (e) there is no significant fragmentation on the heap.

To determine when (in terms of virtual time represented as object allocation count) a garbage collection occurs, we reason about the operation of a lifetime trace-based simulator. A lifetime trace contains a chronological record of all object allocations in an application, along with size and lifetime (number of allocations until an object becomes unreachable) of each object. Using this trace, the simulator allocates objects as directed, and when the combined size of allocated objects reaches the heap size, a garbage collection is triggered. The simulator then discards all unreachable objects (whose lifetime has expired) from the simulated heap. We model this behavior using the following equation:

$$HS = \left(\sum_{j=n_{i-1}+1}^{n_i} SIZE\,[j] \right) + \left(\sum_{\substack{j \in \{1 \ldots n_{i-1}\} \\ DEATH[j] \geq n_{i-1}}} SIZE\,[j] \right) \quad (1)$$

HS is the size of the modeled heap. In real VMs, this corresponds to setting both the minimal and maximal heap size to this value.[1]

n_i is the virtual time of i-th garbage collection. Since the virtual time is measured in object allocations, we know that i-th GC occurred after allocating n_i objects.

$SIZE[j]$ is the size of j-th allocated object in bytes.

$DEATH[j]$ is the virtual time of j-th object's death (object became unreachable). This happens after allocating

[1]Using the -Xmx and -Xms (or similar) parameters.

object number $DEATH[j]$ and before allocating object number $DEATH[j] + 1$. Given the lifetime trace, $DEATH[j] = j + LIFETIME[j]$.

The first term of Eq. 1 thus represents the amount of memory occupied by objects allocated between collections $(i-1)$ and i, while the second term represents the amount of memory occupied by objects surviving the previous $(i-1)$ collections. The whole equation must be understood as an approximation—it is unlikely that the allocated object sizes would exactly add up to the given heap size. However, this particular relaxation simplifies reasoning and makes the equation less complex.

For a given application and heap size, the n_i series is the only unknown in Eq. 1. The values of n_i can be computed with the knowledge of object sizes and lifetimes contained in a lifetime trace, but it requires collecting and processing huge amounts of data.

To make the formula more practical, we replace the exact object sizes and lifetimes by averages, which are easier to obtain. The average object size can be measured by observing the individual allocations. The average object lifetime can be determined indirectly, exploiting the fact that it is necessarily equal to the average number of live objects on the heap, which can be calculated from samples of the number of live objects after each garbage collection. Given the average object size OS and the average lifetime LT, we can simplify Eq. 1 into:

$$n_i - n_{i-1} = \frac{HS}{OS} - LT \qquad (2)$$

The equation then captures an intuitive observation that the average number of objects allocated between consecutive collections (left side) must correspond to the average amount of garbage collected per collection (right side).

4.1 Model Evaluation

Although Eq. 2 is fairly simple, the potential loss of accuracy introduced by averaging is difficult to estimate analytically. We have therefore validated Eq. 2 experimentally for the DaCapo 2006.10 benchmark suite [8] running on the Jikes RVM 3.1.0 with the BaseBaseSemiSpace configuration [3].

For each benchmark, the results in Table 1 list the range of evaluated heap sizes, the average lifetimes, the average object sizes, and the ratio of the measured to the predicted collection intervals (i.e. the number of allocations between collections). A ratio of 1.0 means exact prediction, values greater than 1.0 mean Eq. 2 predicts fewer allocations between collections and vice versa.

Given the extreme simplicity of Eq. 2, we consider the results promising—while not usable for accurate performance prediction, they suffice for better-vs-worse analysis and similar uses. We now proceed with a similar investigation for a common two-generation collector.

5. TWO-GENERATION COLLECTOR

Compared to the one-generation GC discussed earlier, the behavior of a two-generation GC is considerably more complex. We make the following assumptions to build our simplified model: (a) objects have headers and observe address alignment rules, (b) sizes reserved for generations are fixed, (d) the young generation uses copying GC, its memory consists of one eden space and two survivor spaces, (e) the old generation uses mark-and-sweep GC, its memory consists of one old space, (b) GC stops the mutator, (f) minor collection (young generation only) is triggered by full eden space, (g) full collection (both generations) is triggered by close-to-full old space, (h) objects are tenured (promoted from the young to the old generation) after surviving certain number (tenuring threshold) of minor collections, or when a minor collection fills the survivor space, or on a full collection, (i) references pointing from the old to the young generation are in root reference set of minor collections, (j) order of reference traversal is arbitrary, and (k) there is no significant fragmentation on the heap.

Re (b). While generation sizing is usually adaptive, we assume the adaptation to eventually reach a stable state—it is generally recognized that the generation sizes may need to be fixed for optimal performance [16].

Re (d) (e). The choice of a particular type of GC for the young and old generations in our model is not essential—from the modeling perspective, we are mainly interested in the number of memory spaces (and their respective size limits) a particular design uses. We therefore chose to mimic a widely used configuration.

Re (g). The close-to-full condition is modeled by reserving a space in the old generation corresponding to the average size of objects that were promoted during few recent minor collections. The old generation is considered full when the amount of available space drops below this reserve.

Re (j). Order of reference traversal may become important in connection with the tenuring rules. We do not address this aspect due to space constraints.

The above assumptions are a close match for the Serial and Parallel collector configuration found in the HotSpot JVM, and in general fit the GC configuration recommended for maximum throughput in the Oracle HotSpot JVM versions starting with 1.4.

Even after abstracting from the implementation details, the behavior of a two-generation GC remains too complex to hope for useful analogues of Equations 1 and 2—these would turn out to be either overly complex or overly simplified. We therefore proceed by evaluating the model using a simulator.

5.1 Two-Generation GC Simulator

To evaluate the accuracy of our simplified two-generation GC model, we again test the ability of the model to predict the frequency and type (minor or full) of garbage collection cycles. To this end, we have implemented a simulator that takes an application trace, heap configuration, and tenuring threshold as its input and produces a record of all garbage collections triggered during the simulation, including their type and sizes of heap spaces before and after the collection.

The application trace is a more detailed variant of the lifetime trace used for the one-generation GC. Besides object sizes and lifetimes, it also contains records for all reference updates, both in fields and array elements. The heap configuration defines the sizes of the eden and survivor spaces in the young generation, and the size of the old space in the old generation.

During operation, the simulator replays actions from the application trace and keeps track of all objects in all heap spaces, as well as all references that point to objects in the young generation (because such references can make some unreachable objects in the young generation survive minor collections). When a garbage collection is triggered, the simulator performs the appropriate collection and outputs a corresponding collection record.

Table 1: Collection intervals measured / predicted by Eq. 2.

Benchmark	-Xmx, -Xms	LT	OS	Measured / Predicted
antlr	64 – 192 MB	251 293.41 – 266 981.03	65.08 – 65.18	0.88 – 0.95
bloat	128 – 384 MB	459 737.93 – 510 697.55	44.14 – 44.41	0.95 – 1.01
fop	64 – 192 MB	424 407.13 – 483 685.75	48.87 – 49.39	0.94 – 0.96
hsqldb	512 – 1 536 MB	4 182 741.2 – 5 580 740	46.75 – 48.14	0.75 – 0.77
jython	128 – 384 MB	506 214.03 – 506 559.54	69.50 – 69.52	0.79 – 1.00
luindex	64 – 192 MB	239 975.48 – 272 593.07	39.78 – 39.80	0.97 – 1.08
lusearch	128 – 384 MB	281 455.74 – 283 635.68	109.88 – 110.47	1.55 – 1.70
pmd	128 – 384 MB	385 953.04 – 386 825.17	31.35 – 31.36	0.89 – 1.07

5.2 Obtaining Application Traces

There are two basic approaches to obtaining the object lifetime information. The first relies on periodically forcing garbage collection to discover unreachable objects. It is easy to implement but fairly slow and the result accuracy depends on the period between the forced collections. The second approach—based on the Merlin algorithm [13]—is both faster and more accurate, but also more complex and difficult to implement on widely used JVMs.

Because Elephant Tracks [20] (probably the sole currently working implementation of the Merlin algorithm) was not available at the time, and now uses a time metric different from ours, we have developed a tracing tool using DiSL [14] and a custom JVMTI [18] agent. We use the brute force approach to obtain object lifetimes, and always report the granularity (period of forced garbage collections expressed in object allocation units) at which they were collected.

To track object allocations, we instrument the NEW, NEWARRAY, ANEWARRAY, and MULTIANEWARRAY bytecode instructions to report allocation events to the agent, which also receives the VMObjectAlloc events from the JVM. The agent tracks the virtual time (object allocation count) and collects information on object sizes and allocation times. After a specified number of allocations, the agent forces a garbage collection and collects the lifetime information for unreachable objects reported by the JVM via the ObjectFree callback. To track reference updates, we instrument the PUTFIELD and AASTORE bytecode instructions to report reference update events to the agent, which records the new reference and the target it is written to.

5.3 Model Evaluation

To evaluate the accuracy of the model implemented by the GC simulator, we again compare the frequency of young generation and old generation GC cycles reported by the simulator to that observed on a real GC implementation. We perform all experiments on the OpenJDK 1.6.0-22 JVM[2], with heap spaces fixed to predefined sizes and adaptive heap space sizing disabled.[3] This also results in fixing the tenuring threshold at the default value of seven.

We collect the application traces for selected workloads from the DaCapo 9.12-bach benchmark suite [8]—here, we report specifically on the batik, fop, xalan and tomcat workloads. Given that these are fixed-duration benchmarks, the evaluation metric can be simplified to the number of garbage collection cycles.

Because all the workloads have relatively modest memory footprints, we iterate over each workload 100 times.[4] To provide an alternative scaling method, we implemented a modified benchmark harness that executes multiple copies of the same workload in parallel and uses multiple class loaders and separate data directories to isolate the executing workload instances. Using this harness, we run the workloads in 8 threads, iterating over each workload 10 times in each thread. Due to various technical issues, this scaling method works reliably only with the fop workload, which we refer to as multifop.

Limiting the spectrum of the benchmark workloads was motivated by different factors for each workload. The eclipse, tradebeans and tradesoap workloads use class loading in a manner that is not compatible with the code instrumentation required by our experiments. The avrora, lusearch and luindex workloads do not exhibit interesting behavior with respect to garbage collection frequencies. The h2 and jython workloads generate an excessively large trace that our infrastructure was not able to accommodate.

We should point out that despite omitting some workloads, the range of experiments we perform is still extreme—a single set of traces from the selected workloads is close to quarter of a terabyte in size. Just collecting such a set takes over a month of parallel execution time on a 2.33 GHz eight-core machine, and the time to simulate the considered heap size configurations for a single workload—a single line in some of the plots presented later—is measured in days.

We first report the results obtained when providing the simulator with a complete application trace, which includes object lifetimes, sizes, and reference updates.

For the baseline evaluation, the application traces were collected with the following granularities: 10000 allocations for batik and fop, 2000 allocations for multifop and tomcat, and 1000 allocations for xalan. These choices help maintain variability between the experiments while balancing accuracy and overhead.

For the heap size configuration, we use a combination of 8 young generation sizes and 6 old generation sizes, yielding 48 heap size configurations for each benchmark. The range of young generation sizes is the same for all benchmarks: 16, 24, 32, 48, 64, 96, 128, and 192 MB. The size of each of the two survivor spaces is always 1/8 of the young generation

[2]OpenJDK Runtime Environment IcedTea 6 1.10.3 Gentoo Build 1.6.0-22-b22 and OpenJDK 64-Bit Server VM Build 20.0-b11 Mixed Mode

[3]Using the -XX:ParallelGCThreads=1 -XX:-UsePSAdaptiveSurvivorSizePolicy -XX:NewSize -XX:MaxNewSize -Xmx -Xms JVM options.

[4]Using the DaCapo -no-pre-iteration-gc -n100 options.

size, leaving the remaining 6/8 for the eden space. The range of old generation sizes is given in the following table—the benchmarks differ in memory requirements, we therefore choose the ranges so that the smallest size in the range is always close to the bare minimum required to execute the benchmark.

Due to space limitations, we plot results from this and the following two sections together in Figures 1–12. The legend to the plot labels is in Table 2, the first four labels are relevant to this section. While this arrangement makes it difficult to discern individual results, it fits the goal of illustrating the differences between results of various experimental configurations in the limited space of the paper.

Benchmark	Old generation sizes (MB)					
batik	128	160	192	256	384	512
fop	64	128	192	256	384	512
multifop	256	288	320	384	512	1024
tomcat	48	64	96	128	192	256
xalan	160	192	256	384	512	768

Table 2: Plot legend labels

Legend label	Configuration
JVM: JIT	JVM in default mode with JIT enabled
JVM: no JIT	JVM in interpreted mode (-Xint option)
JVM: DiSL	JVM with instrumented code
Default	Simulator with complete input
P(survived)	Simulator with lifetime trace and probability of object being marked and because of that surviving
P(marked)	Simulator with lifetime trace and probability of object being marked
LT&SZ only	Simulator with lifetime trace and object sizes only
Generated 1	Simulator with generated lifetime trace, seed 1
Generated 2	Simulator with generated lifetime trace, seed 2

The plots in Fig. 1 and in Fig. 2 show the young generation GC counts for the fop and multifop workloads, respectively. The results obtained from the GC model simulator with full application trace as an input are labeled *Default*, while the results observed on a real GC are labeled *JVM: JIT*. We omit plots for other workloads to save space, because all result variants are very similar to fop, and in general show good accuracy with the exception of the multifop workload.

The minor collection counts for the simulated and the real GC should approximately equal the total size of all allocated objects divided by the eden size. The large difference between the simulated and the observed collection counts therefore indicates that the total sizes of objects observed during the trace collection and during the actual JVM execution differ. The reason for the difference rests with the escape analysis performed by the JIT. It is used to introduce stack allocation for objects that only exist in the scope of one method. Because our instrumentation calls a native method with a newly allocated object as a parameter, it makes all object escape and thus effectively disables the stack allocation.

Until we can include this optimization in the GC model, we can disable it by running the benchmarks with the tracing instrumentation inserted (even when no agent is using it). In the plots, the results of this configuration are labeled *JVM: DiSL*. The minor collection counts from the simulator then become very similar to the counts observed in the instrumented JVM. As a sanity check, we also execute the benchmarks in interpreted mode, with results labeled *JVM: no JIT* in the plots. We should point out that this particular issue is likely to impact all tools that rely on instrumentation to collect traces, even when those tools claim to be precise, such as Elephant Tracks [20].

The plots in Figures 3, 5, 7, 9 and 11 show the full GC counts for the batik, fop, tomcat, xalan, and multifop workloads, respectively. The results from the GC model simulator obtained using complete application trace are labeled *Default*, while the results from a real GC observed in three JVM runs—default, instrumented, and interpreted—are labeled *JVM: JIT*, *JVM: DiSL*, and *JVM: no JIT*, respectively. Depending on the workload and heap configuration, the prediction accuracy varies from very high (fop in smaller heap) to very low (tomcat in smaller heap). The plots alone contribute to our goal of illustrating how far a simplified model explains a real GC implementation. We analyze some reasons in more detail in the following sections.

6. IMPACT OF REDUCED INPUT

In this section we complement the baseline evaluation with experiments that focus on finding the limits and trends in model accuracy depending on the available input data. Complete application trace—lifetimes, object sizes and reference updates—are huge, easily into gigabytes for workloads that only take a few minutes to execute. Collecting such traces is neither always possible nor always practical, and simulation with complete input data is also computationally expensive—merely reading the input data usually takes longer than executing the workloads. It is therefore important to understand what accuracy can be expected when some of the input data is aggregated or approximated, which is an approach any practical models would have to follow.

In Sect. 6.1, we approximate the reference updates information in the input data with a single probability value, leaving only the lifetimes and object sizes. In Sect. 6.2, we experiment with ignoring the reference updates altogether. And finally, in Sect. 6.3, we also replace object lifetimes and sizes with probability distributions.

6.1 Lifetime Trace with Mark Probabilities

This is the experiment where we start to shrink the model input data. We start with the reference update trace, which usually makes up more than 80 % of the input size. Our goal is to discard this data but still model the fact that some unreachable objects in the young generation survive minor collections due to references from the old generation.

Our approach is to replace the reference update trace with one of two stochastic approximations. We compute the probability that an object is reachable from the old generation—marked for short—during a minor garbage collection, either for all objects (denoted *P(marked)*) or for objects whose lifetime has expired (denoted *P(survived)*). For illustration, we show the probabilities for fop in Fig. 13 and for tomcat in Fig. 14. Both probabilities are relatively stable for a given young generation size across all our benchmarks, we therefore evaluate our model with one value of *P(marked)* and one value of *P(survived)* for each young generation size.

We calculate the average probabilities for each heap configuration using our simulator—we were hoping to approximate

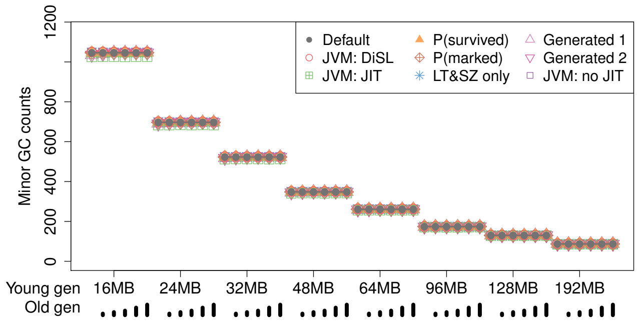


Figure 1: Young GC counts: fop

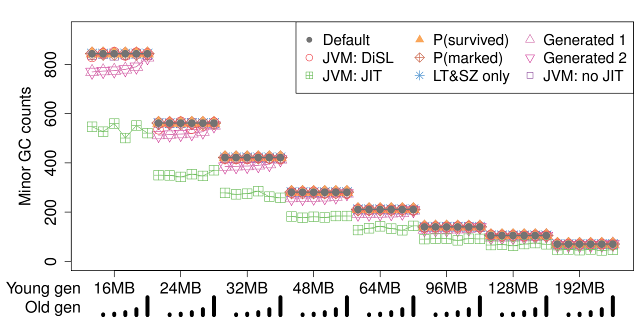


Figure 2: Young GC counts: multifop

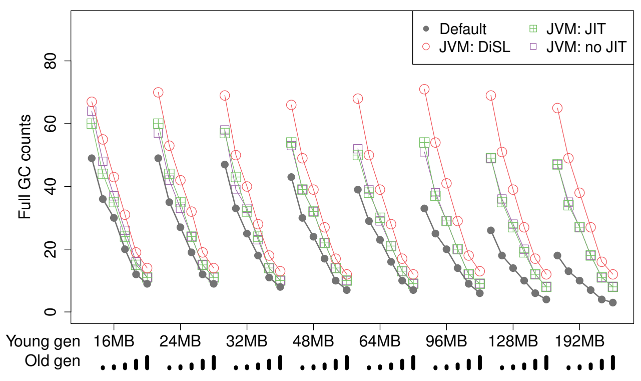


Figure 3: Full GC counts – JVM: batik

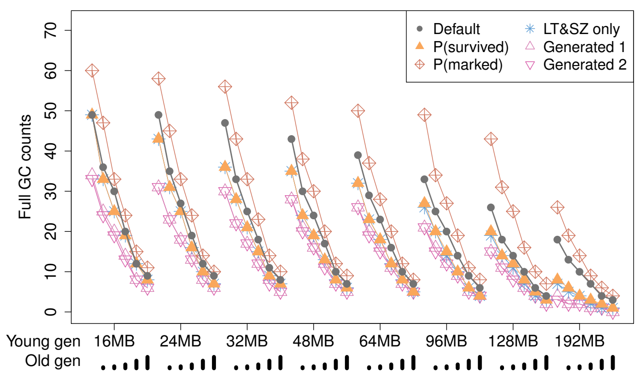


Figure 4: Full GC counts – simulators: batik

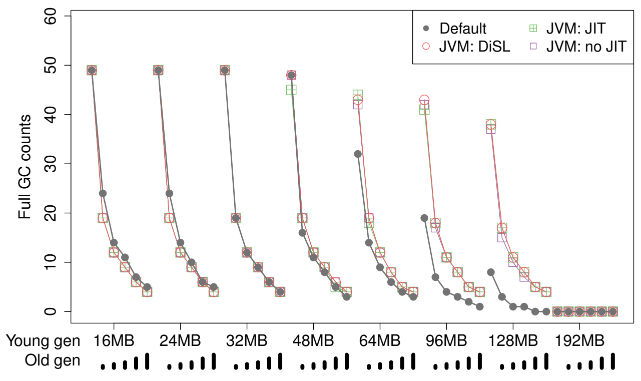


Figure 5: Full GC counts – JVM: fop

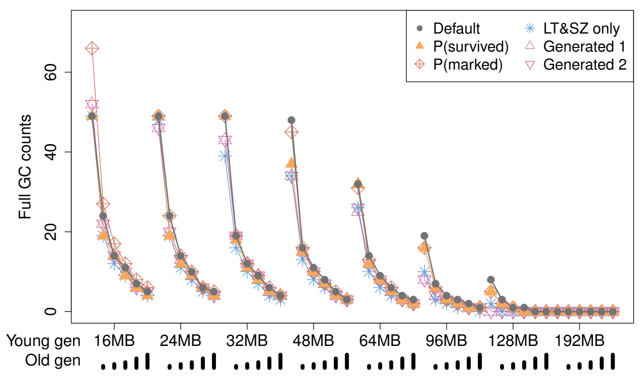


Figure 6: Full GC counts – simulators: fop

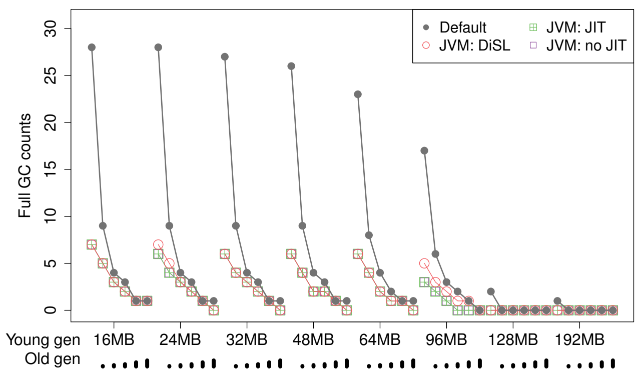


Figure 7: Full GC counts – JVM: tomcat

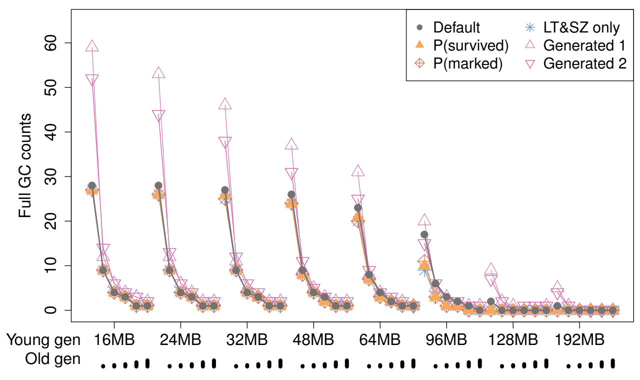


Figure 8: Full GC counts – simulators: tomcat

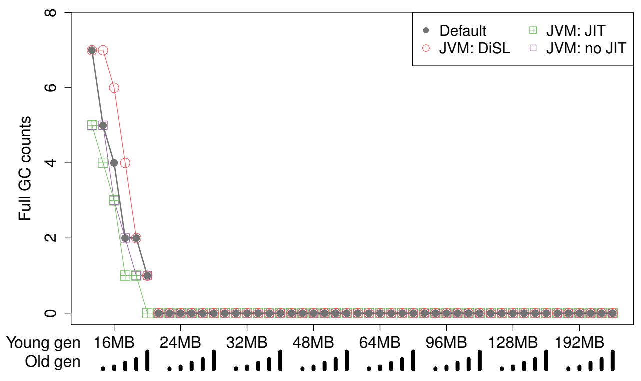


Figure 9: Full GC counts – JVM: xalan

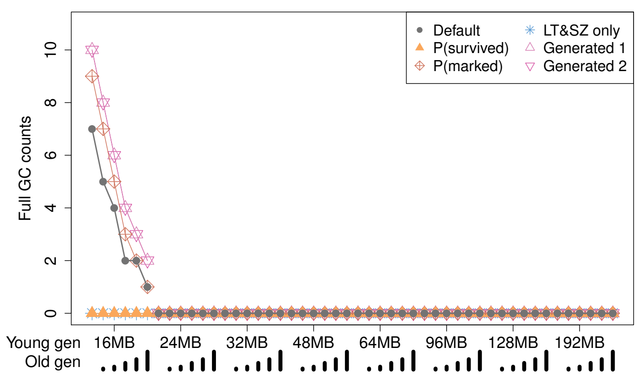


Figure 10: Full GC counts – simulators: xalan

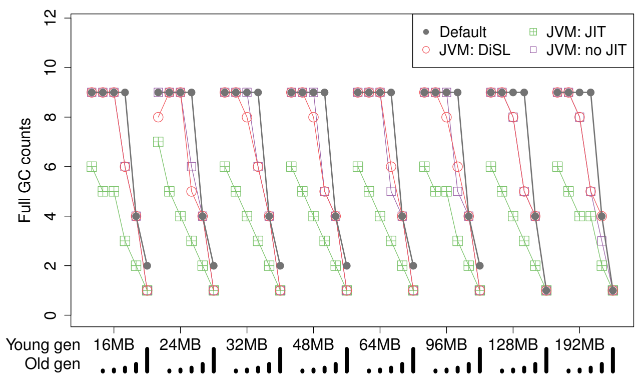


Figure 11: Full GC counts – JVM: multifop

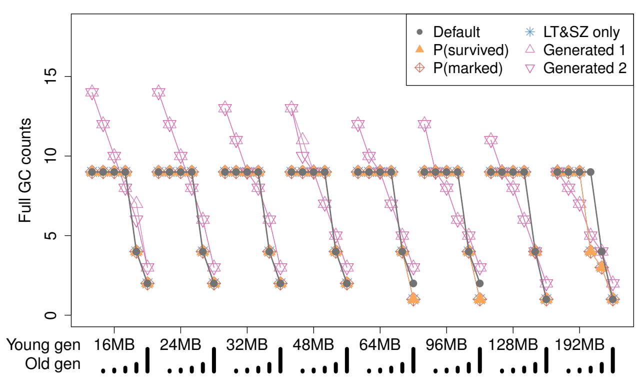


Figure 12: Full GC counts – simulators: multifop

the probabilities from some benchmark or configuration characteristic, but we have not found a way to do so.

When using the *P(marked)* probability, the simulator randomly marks all objects in the young generation every minor collection, with probability *P(marked)*. When using the *P(survived)* probability, the simulator randomly marks only those objects whose lifetime has expired, with probability *P(survived)*. In both cases, the marked objects survive the minor collection regardless of their actual lifetime.

The results from the experiments described in this section are in Figures 4, 6, 8, 10 and 12. The results labeled *Default* are from simulations with complete input, the results labeled *P(marked)* and *P(survived)* are from simulations that respectively use one of the two probabilities.

6.2 Lifetime Trace Only

In this set of experiments, we completely avoid reference updates and use only lifetime and size of objects. This means that no unreachable objects survive the simulated collection—both minor and full collections are complete. The number of minor collections should not change, the number of full collections can be smaller than in the previous experiments—this is confirmed in Figures 4, 6, 8, 10 and 12, where the results from this experiment are labeled *LT&SZ only*.

6.3 Lifetime and Size Distributions

The last set of experiments uses the smallest input, replacing the entire trace with a table that tells the probability of records with particular lifetime and object size appearing in the trace. The table consists of buckets that correspond to lifetime ranges, each bucket lists unique object sizes and counts for objects with that lifetime. In addition to the table, which characterizes lifetimes and object sizes, we use the *P(survived)* probability from Sect. 6.1.

The lifetime ranges are used to keep the table reasonably small, however, we have to be careful to avoid losing too much information. Accuracy is essential for objects with small lifetimes, where fluctuations influence the tenuring decision, and for objects with large lifetimes, where fluctuations influence average old generation occupancy. In contrast, knowing medium lifetimes accurately is of smaller importance. We use tables of 200 buckets, with eight lifetime ranges for the smallest lifetimes and five lifetime ranges for the largest lifetimes growing and shrinking in logarithmic steps, the ranges of the remaining buckets are of equal size.

The buckets keep exact sizes and counts. Our benchmarks use only about 500 to 1200 different object sizes, which makes keeping exact sizes possible. For workloads that generate objects of many different sizes (for example arrays with varying sizes), we would modify the algorithm to create size buckets as well.

To avoid potentially error-prone modifications, we keep our simulator as is and run it on synthetic traces that conform to the description in the table—that is, we first compute the tables that characterize our benchmarks and then simulate GC on traces generated from these tables. The procedure of generating such traces is described next.

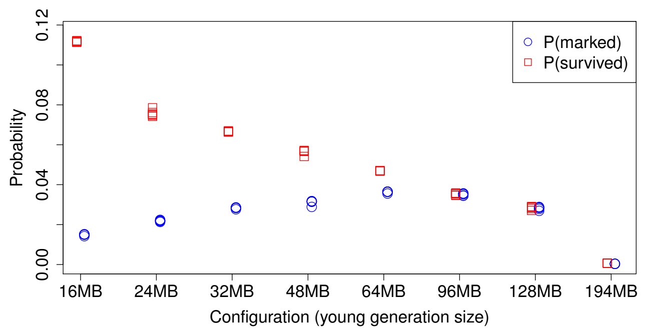


Figure 13: Mark probabilities: fop

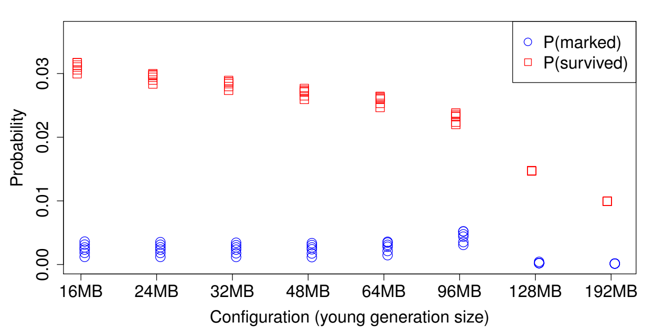


Figure 14: Mark probabilities: tomcat

Trace Generator Description.

The procedure of generating a trace from the table of lifetimes and sizes is complicated by the fact that individual lifetimes are not independent random variables—in particular, when there are only N allocation events left to generate in the trace, the biggest lifetime the allocated object can have is also N.

Our trace generation algorithm addresses the problem as follows. At any moment, we know the number of allocation events still to be generated (denoted N), the bucket whose lifetime range includes N (here called the oldest bucket), and the number of objects to be generated from the oldest bucket (denoted I). For the oldest bucket, we prepare I random lifetimes in the corresponding lifetime range, sorted by value. When N is greater than the oldest prepared lifetime, we pick a random bucket and a random size from that bucket and emit a corresponding allocation event into the generated trace. When N reaches the oldest prepared lifetime in the oldest bucket, we pick a random size from that bucket and emit an allocation event with the oldest prepared lifetime and the chosen size. After emitting an event, we decrement N (this may designate new bucket as the oldest bucket), decrement the count of objects of the used size in the used bucket, and remove the prepared lifetime from the oldest bucket if applicable.

The random bucket choice uses a discrete probability distribution, the probability of picking a bucket corresponds to the share of objects to be generated from the bucket. The random lifetimes are picked from a uniform distribution with minimum and maximum corresponding to the lifetime range of the bucket. For practical reasons, we do not prepare the random lifetimes for the buckets with shortest lifetimes.

We present results of two simulations for each benchmark. The input was created using the trace generation algorithm with two different random number generator seeds. The results are displayed in Figures 4, 6, 8, 10 and 12. We use the legend labels *Generated 1* and *Generated 2* for the data.

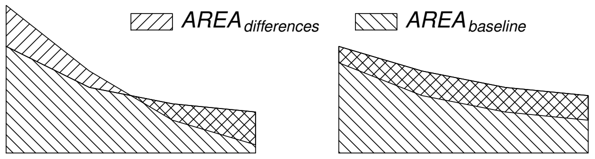
6.4 Accuracy Metric

Besides the visual evaluation using the plots in Figures 5–12, we also provide a numeric accuracy metric. Among typical model evaluation metrics are the ratio of the model results to the measured values, or the proportion of successful predictions (i.e. results within tolerance) to all predictions.

In our case, such metrics would allow reporting arbitrarily good accuracy by including more configurations where no collections happen—as in the xalan workload. We therefore use a metric based on the relative area difference in the plots, which eliminates the effect of configurations with no collections. We denote this metric as inaccuracy, calculated as follows:

$$Inaccuracy = \frac{AREA_{differences}}{AREA_{baseline}} \tag{3}$$

The $AREA_{differences}$ is the area between the two lines of plots we compare and $AREA_{baseline}$ is the area under the plot depicting the baseline—the two areas, which can overlap, are shown on the following illustration.



For the scale, we use collection count on the vertical axis and equidistant units on the horizontal axis. For the full collections, the results are shown in Table 3. We use the instrumented JVM runs as the baseline. The smaller the value is, the better the accuracy—zero means perfect fit.

The table shows that although the results across benchmarks fluctuate, the overall tendency is a gradual decrease in accuracy as the inputs are reduced. As an anomaly, the accuracy with the reduced input based on *P(marked)* appears better than the accuracy with full input. This is due to the fact that using *P(marked)* leads to overestimating the number of objects surviving young collections, and because the model with full input tends to predict fewer collections, this overestimation turns out to be helpful.

Table 3: Inaccuracy for full collections

Simulator	batik	fop	multifop	tomcat	xalan
Default	0.46	0.28	0.14	1.31	0.26
P(marked)	0.30	0.35	0.13	1.09	0.13
P(survived)	0.57	0.30	0.13	1.10	1.00
LT&SZ only	0.57	0.38	0.13	1.08	1.00
Generated 1	0.66	0.36	0.23	2.39	0.17
Generated 2	0.67	0.36	0.22	2.26	0.17

Overall: *Default* 0.41, *P(marked)* 0.32, *P(survived)* 0.48, *LT&SZ only* 0.50, *Generated* 0.60

6.5 Results Discussion

From the results presented on the GC count plots (Figures 1–12), we can tell that the simulation gives accurate counts of minor collections, but the accuracy of the full collection counts is limited. This is mostly an expected result, because our simplified model does not capture all the behavior of the JVM collector implementation and because the input trace is not precise—we illustrate the sensitivity to inputs in Sect. 7.

The good accuracy in predicting minor collections is related to the simplicity of the triggering condition. The matching results confirm that the total size of the objects in the trace is roughly the same as in the real application run. We have observed only one exception (especially visible in the multifop workload), which we attribute to the use of escape analysis for stack allocation. We have separated the effect in evaluation by using the instrumented JVM runs as the baseline.

Another optimization that could affect the accuracy is the usage of Thread-Local Allocation Buffers (TLAB)—small memory buffers the threads allocate from to minimize locking. Among our workloads, xalan, tomcat and multifop use more mutator threads, but the results show no anomalies, we therefore conclude that TLAB use does not affect the minor collection count considerably.

Restricting the input data sets cannot impact the predicted minor collection count except for the randomly generated traces. In the other experiments, the total size of objects in the traces does not change and therefore the minor collection counts must remain the same as well. For the generated traces, some differences may occur in principle, but our results show they are small—the total inaccuracy in predicting minor collections in the two simulations with generated traces is 0.069 and 0.072, almost the same as in the simulations with measured traces (0.068 across all traces).

When it comes to the full collection counts, we can summarize the results as follows: good accuracy for multifop and xalan, often but not always good accuracy for fop, poor accuracy for batik and tomcat workloads. This summary is for the JVM runs with instrumentation enabled, which isolates the escape analysis issue.

One reason for the poor accuracy cases rests with the trace collection method, as analyzed in [13]. For a particular tracing granularity, the collected lifetimes increase on average by half the granularity value. This increase is reflected in larger live heap sizes and should therefore cause more collections. This is not what we observe, however—when inaccurate, the simulator tends to predict fewer full collections. This suggests our trace collection method is not the (sole) cause of the result inaccuracy.

As an important observation, we note that the full GC counts are fractions of 100 ($100/X - 1$ for various X, i.e. 99, 49, 32, 24) for a surprisingly large number of heap configurations. Given that we use 100 iterations in the DaCapo workloads, this is unlikely to be a coincidence. We illustrate the effect in detail in Fig. 15, where we show the full collection counts for fop across more heap configurations—the old generation sizes are 44, 48, 52, 56, 60, 64, 72, 80, 88, 96, 112, 128, 160, 192, 224, 256, 320 and 384 MB. The data points would normally roughly follow the $1/x$ hyperbolic shape, as is the case for the 128 MB young generation size, but the results show clusters at 49, 32 and 24 collections (emphasized by dotted lines in the plot) for the other three young generation sizes.

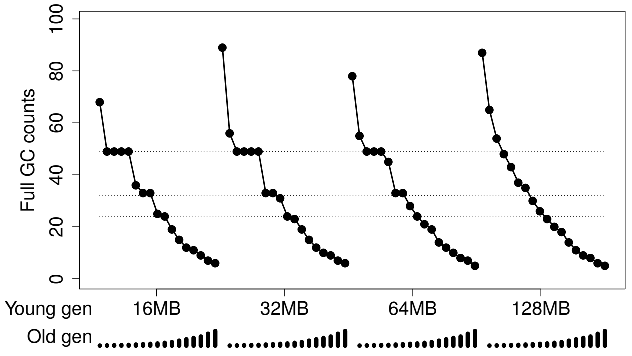


Figure 15: Dense configurations: fop

To explain this phenomenon, we look into how the live size of the workloads changes in time. Using the allocation count as the time unit, we plot the live heap sizes calculated from the traces of the batik, fop, multifop and tomcat workloads. Figures 16 and 18 show the first four iterations out of 100 for fop and batik, Fig. 17 the first four out of 10 for multifop, and Fig. 19 the first nine out of 100 for tomcat. The sawtooth shape suggests all four workloads release most of the objects allocated in each iteration. Additionally, the gradual rise between iterations in tomcat resembles a memory leak.

The sawtooth shape is due to the way the DaCapo harness implements iterations. Most of the objects allocated in an iteration become garbage at the iteration end. This makes the minimum memory requirements of the workload (minimum heap size where the workload still executes) close to the minimum memory requirements of a single iteration. Each new iteration will allocate new objects and unless the heap size exceeds the minimum requirements at least twice, GC will be triggered. This GC will release objects from the past iterations (which since became garbage), providing enough memory for this iteration but not the next one, and the entire cycle will repeat. As a result, the number of collections will match the number of iterations for any heap size between the minimum requirements and twice the minimum requirements. Along the same lines, the number of collections will be half the number of iterations if two but not three iterations fit the heap size, and so on. This explains the clusters in Fig. 15.

The sawtooth shape not only makes the workload less sensitive to heap size changes, it also makes the GC more difficult to predict. Clearly, a GC cycle triggered just before the end of an iteration will free much less memory than a GC cycle triggered just after the end of an iteration, even though the two can be just a few allocations apart. The impact on GC count can be large because the former situation will require another GC sooner rather than later, and there is no guarantee the new GC will be more successful. As an example of this effect, the batik workload (configuration with 16 MB young and 128 MB old generation) triggers full garbage collections with the live sizes of 60 MB in the instrumented JVM and 40 MB in the default simulator. This is a major factor for the prediction accuracy results we observe.

Our analysis is further supported by the difference in results between the default simulator and the simulation with generated traces—the traces generated from the tables do not exhibit the sawtooth shape of the live heap size, making the clusters disappear. This is very visible on the results for the

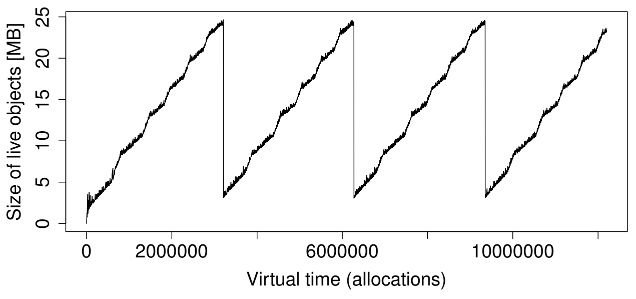


Figure 16: Partial live size trace: fop

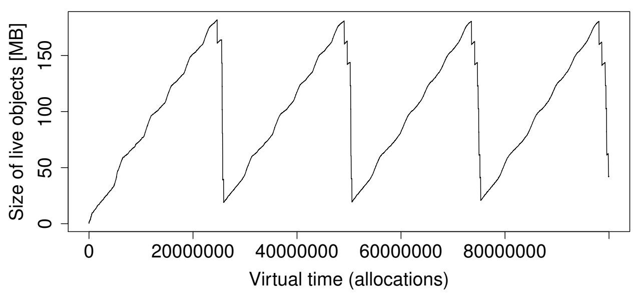


Figure 17: Partial live size trace: multifop

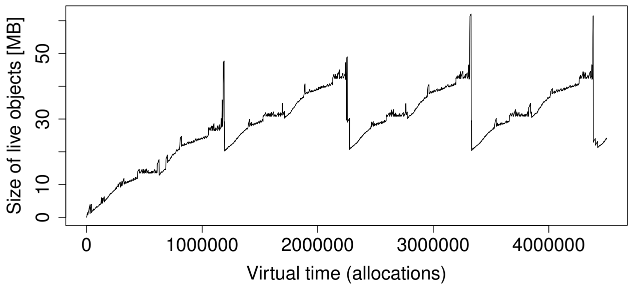


Figure 18: Partial live size trace: batik

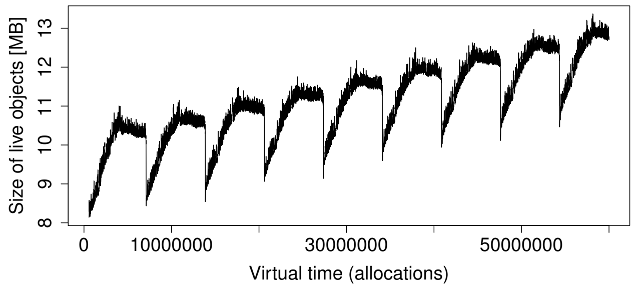


Figure 19: Partial live size trace: tomcat

multifop workload (Fig. 12), where horizontal clusters evident when simulating real traces change into gradual slopes with the generated traces.

Finally, the gradual rise in the live heap size between iterations in tomcat also complicates predictions. As the heap becomes more and more occupied, the GC frequency increases and any loss of accuracy is magnified.

7. IMPACT OF INACCURATE INPUT

Collecting inputs for the GC model simulator is non-trivial and not always guaranteed to provide fully accurate information—this is most pronounced in the case of object lifetimes, where the collection granularity directly influences lifetime accuracy (see Sect. 5.2). Technically, the inaccuracy due to data collection process results in different values for object sizes and lifetimes in the input data—we can as well modify the input data ourselves to determine how certain changes in the workload, e.g. systematically allocating more objects or enlarging object sizes, influence the GC behavior.

We therefore perform sensitivity analysis to determine how certain changes in object lifetimes and sizes impact the model results. For object lifetimes, we consider changes due to an additive constant, a multiplicative factor, a random error, and limits on the minimum and maximum lifetimes. For object sizes, we consider changes due to an additive constant, a multiplicative factor, and a random error. Due to space limitations, we only report results for changes due to a multiplicative factor, and a random error.

7.1 Sensitivity to Lifetime Changes

Multiplying object lifetimes by a constant factor models two hypothetical situations. In the first, a process collecting lifetime information systematically ignores certain allocations, perhaps because it could not instrument all paths in the JVM that allocate objects. In the other, we may be interested in what happens if a certain workload started to systematically allocate more objects with short lifetimes. In both cases, if either the missing or additional allocations were spread evenly throughout the workload, it would correspond to scaling all object lifetimes.

Figure 20 shows how the number of full collections changes when all lifetimes are scaled using a constant factor, i.e. $l' = l \times k$ for chosen values of k. For $k < 1$, the object lifetimes are shortened, and the resulting trace may contain reference updates on objects whose lifetime has already expired—we remove such invalid reference updates from the trace. We only investigate the impact on the number of full collections, because minor collections are lifetime insensitive.

The results illustrate how lifetime scaling interacts with the young generation size. For a young generation that is small relative to the workload requirements (16 MB), the effect of scaling the lifetimes down is subdued—most objects still live long enough to be promoted and cause full collections. For a young generation that is large relative to the workload requirements (64 MB), it is the effect of scaling the lifetime up that is subdued—most objects that die young before scaling also die young afterwards.

Adding a random error to object lifetimes models the effect of collecting lifetimes with a particular collection granularity, which necessarily impacts our experiments (c.f. Sect. 5.2). To include both frequent small deviations and occassional large ones, we model the error as a random variable with a shifted exponential distribution, the observations of which are added the object lifetimes, i.e. $l' = l + Exp(1/\mu) - \mu$ for chosen values of μ. We adjust the possibly negative lifetimes so that the modification preserves the average lifetime.

The results for selected values of μ are shown in Fig. 21. The observed effects are again related to the young generation size—the average object size in fop is 95 B, a young generation of 16 MB can accommodate about 155000 such objects, an object therefore has to live at least around million allocations

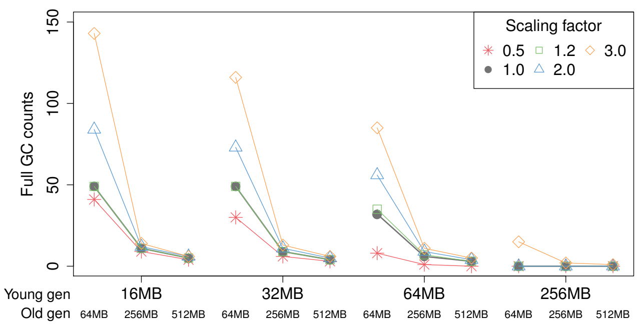


Figure 20: Lifetime scaling: fop

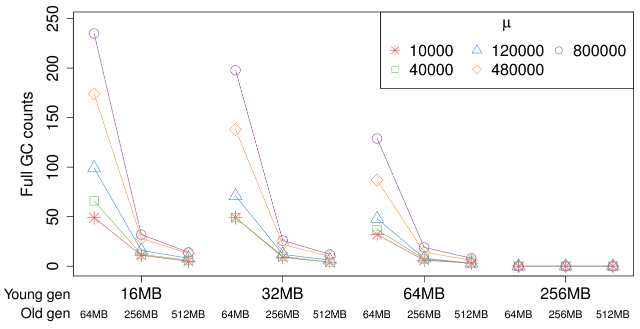


Figure 21: Lifetime randomization: fop

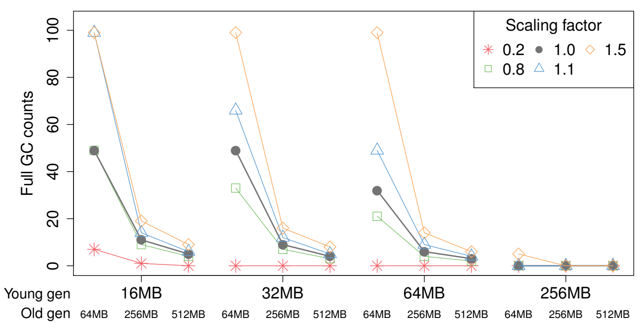


Figure 22: Size scaling: fop

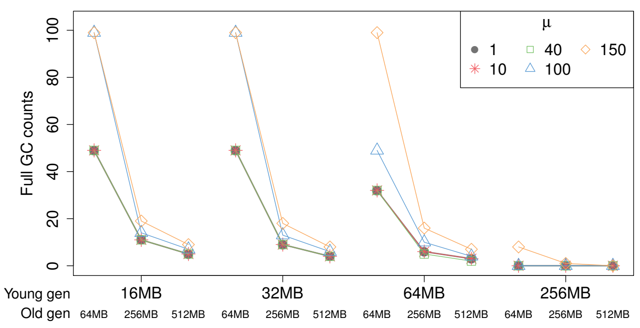


Figure 23: Size randomization: fop

to be tenured. With larger young generations, the numbers grow further, making it less likely that the positive random errors get enough objects tenured to impact the number of full collections. The negative random errors in our experiment are bounded by μ and therefore even less significant than the positive ones.

7.2 Sensitivity to Object Size Changes

Multiplying objects sizes by a constant factors models a situation where we change the size of a fundamental data type that is used by most objects, e.g. by introducing compressed references [1] to reduce memory overhead on 64-bit systems.

Figure 22 shows how the number of full collections changes when all object sizes are scaled using a constant factor, that is, $s' = s \times k$ for chosen values of k. The results again highlight the clustering effect discussed in Sect. 6.5—for the young generation size of 16 MB, deflating all objects by 20 % has no effect, and inflating all objects by 10 % has the same effect as inflating by 50 %.

Adding a random error to object sizes again models the inaccuracies we may encounter when collecting application traces, e.g. a systematic measurement error due to object size alignment rules. Again, we model the error as a random variable with a shifted exponential distribution, the observations of which are added the object sizes, that is, $s' = s + Exp(1/\mu) - \mu$ for chosen values of μ.

The results for selected values of μ are shown in Fig. 23. They again confirm the clustering effect and show that it is not sensitive to small changes in object size.

8. CONCLUSION

Motivated by the need to understand garbage collection behavior from the application developer perspective, and some motivating results from one-generation GC, our work uses extensive experiments to compare the behavior of a real GC implementation with the behavior of a simplified model, such as the developer may form based on commonly available information [19, 21].

Given an almost-complete information about workload behavior in the form of application traces with object sizes, lifetimes, and reference updates, we show that the model can fairly accurately predict frequency of minor garbage collections in a two-generation GC.

The model retains a relatively stable prediction quality across workloads and inputs ranging from full application traces to probabilistic distributions of object sizes and lifetimes. However, predicting the frequency of full collections for the very same two-generation GC turns out to be a very different story—even with full application trace used as the simulator input, the prediction quality is mediocre, ranging from 14 % inaccuracy to 131 % inaccuracy in our examples.

We illustrate how the prediction quality gradually deteriorates as the inputs of the model are reduced. The overall tendency is a gradual decrease, from 41 % inaccuracy to 60 % inaccuracy in our metric. Looking at the individual workloads, the inaccuracy could be much worse, exceeding 200 % in case of tomcat.

The prediction quality ultimately depends on the ability of the GC model to accurately evaluate the GC triggering conditions. In the case of the full collections, this seems to be particularly difficult, because small changes in the input or in the interactions among detailed features can significantly impact the observed behavior. In our experiments, we have seen how reducing object size by 20 % did not impact full collection count at all, or how increasing object size by 10 %

doubled the full collection count, but further increase by 40 % did not have an impact anymore.

This is unfortunate from the developer perspective, who would naturally expect a reasonable reaction to workload changes. While we explain the causes for such behavior when analyzing the results, we were only able to do that with detailed insight, which goes beyond the basic principles our GC model is built with. Therefore, besides illustrating the complex character of interactions that govern the behavior of contemporary garbage collectors, our work also explains why—rather than getting definite instructions on garbage collector configuration—application developers are instead given recommendations for trial-and-error tuning.

Our experiments are also related to the available knowledge about sensitivity to workload parameters. Earlier work [13] points out that exact knowledge of object lifetimes is important for accurate simulation of several garbage collector metrics including ratio of live to allocated objects or number of reference updates that cross generation boundaries. We illustrate the sensitivity to lifetime changes and object size changes on the simplified model.

Finally, our experiments draw attention to drawbacks of the existing garbage collector evaluation methods. One concerns the process of collecting the workload traces—we highlight how program instrumentation interferes with the escape analysis, effectively disabling a class of stack allocation optimizations. This makes it possible to better qualify the behavior of tools that use instrumentation to collect the workload traces, such as Elephant Tracks [20]. While such tools may collect an accurate trace of the allocation operations in the application, this is not necessarily an accurate trace of the operations that manipulate the heap.

The final issue concerns the behavior of the workload scaling method in the DaCapo benchmark suite [8]. The repetition of isolated workload instances creates memory usage profiles that regularly make most objects unreachable, leading to possibly anomalous situations where changes in the heap size have no impact on the garbage collection frequency.

To complement our submission, complete tools and results are available on-line at http://d3s.mff.cuni.cz/papers/gc-modeling-icpe.

Acknowledgements

This work was partially supported by the Czech Science Foundation project GACR P202/10/J042, the EU project ASCENS 257414, the Swiss National Science Foundation project CRSII2_136225, and the Charles University institutional funding.

9. REFERENCES

[1] A. Adl-Tabatabai et al. Improving 64-bit Java IPF performance by compressing heap references. In *CGO*, 2004.

[2] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM*, 2010.

[3] B. Alpern et al. The jalapeño virtual machine. *IBM Syst. J.*, 39(1), Jan. 2000.

[4] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. In *ISMM*, 2010.

[5] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1), 2009.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *Perform. Eval. Rev.*, 32(1), 2004.

[7] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, 2008.

[8] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10), 2006.

[9] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

[10] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[11] S. Joshi and V. Liaskovitis. *Java Garbage Collection Characteristics and Tuning Guidelines for Apache Hadoop TeraSort Workload*, 2010.

[12] P. Libič, P. Tůma, and L. Bulej. Issues in performance modeling of applications with garbage collection. In *QUASOSS*, 2009.

[13] M. Hertz et al. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3), May 2006.

[14] L. Marek, Y. Zheng, D. Ansaloni, W. Binder, Z. Qi, and P. Tuma. DiSL: an extensible language for efficient and comprehensive dynamic program analysis. In *DSAL*, 2012.

[15] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM*, 2008.

[16] Oracle. *Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning*. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html.

[17] Oracle. *Tuning Garbage Collection with the 5.0 Java Virtual Machine*, 2003. http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html.

[18] Oracle. *JavaTM Virtual Machine Tool Interface*, 2011. http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/.

[19] T. Printezis. *Garbage Collection in the Java HotSpot Virtual Machine*, 2004. http://www.devx.com/Java/Article/21977.

[20] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: portable production of complete and precise gc traces. In *ISMM*, 2013.

[21] Sun Microsystems, Inc. Memory management in the Java HotSpot virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006.

[22] K. Ueno, A. Ohori, and T. Otomo. An efficient non-moving garbage collector for functional languages. In *ICFP*, 2011.

[23] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *ISMM*, 2009.

[24] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *ISMM*, 2013.

[25] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy. Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes*, 31(2), 2006.