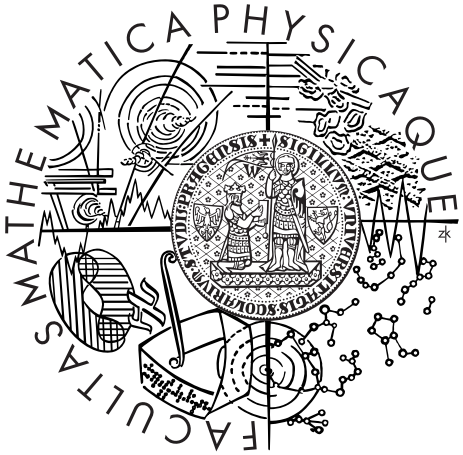
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Peter Libič

Garbage Collection in Software Performance Engineering

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis: doc. Ing. Petr Tůma, Dr.
Study program: Computer Science
Specialization: I2 Software Systems

Prague 2015

I would like to thank all colleagues from the department for creating a friendly and supporting atmosphere so that it was a pleasure to work with them. I want to thank especially my advisor Petr Tůma for his patience and much appreciated help with this thesis and being nice to me all those years, even if I didn't deserve it on many occasions. I also want to thank my fellow PhD student Vojtěch Horký, who keeps helping me all the time, for those beautiful R plots and for all the fun we had. I also want to apologize to everyone for eating their food. I express my gratitude to Matthias Hauswirth and Thomas Würthinger who let me work in their teams.

I thank all my close friends for their much appreciated support. Last but not least, I want to thank my family for everything they are doing, were doing and will be doing for me.

This work and related research was partially supported by the EU project Q-ImPrESS, the EU project ASCENS, Charles University institutional funding (SVV-2015-260222, SVV-2014-260100 and GAUK Project 156810), Swiss Sciex project 09.042 and by the Grant Agency of the Czech Republic project GACR P202/10/J042.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 28, 2015 . . . . . .

Annotation

Title	Garbage Collection in Software Performance Engineering
Author	Peter Libič
	peter.libic@d3s.mff.cuni.cz
Advisor	doc. Ing. Petr Tůma, Dr.
	petr.tuma@d3s.mff.cuni.cz
Department	Department of Distributed and Dependable Systems
	Faculty of Mathematics and Physics
	Charles University
	Malostranské nám. 25, 118 00 Prague, Czech Republic

Abstract

The increasing popularity of languages with automatic memory management makes the garbage collector (GC) performance key to effective application execution. Unfortunately, performance behavior of contemporary GC is not well understood by the application developers and often ignored by the performance model designers.

In this thesis, we (1) evaluate nature of GC overhead with respect to its effect on accuracy of performance models. We assess the possibility to model GC overhead as a black-box and identify workload characteristics that contribute to GC performance. Then we (2) design an analytical model of one-generation collector and a simulation model of both one-generation and two-generation collectors. These models rely on application characteristics. We evaluate the accuracy of such models and perform an analysis of their sensitivity to the inputs. Using the model we expose the gap between understanding the GC overhead based on knowing the algorithm and the actual implementation. In the course of evaluation we discover important GC issues concerning application developer and suggest how to tackle those issues. Last, we (3) design a model to help the developer predict effects of certain code additions on GC overhead of the whole application. This model is easy to use and only uses readily obtainable inputs and workload characteristics.

Keywords

Garbage collection, performance, modeling, Java

Anotace

Název	Garbage Collection in Software Performance Engineering
Autor	Peter Libič
	peter.libic@d3s.mff.cuni.cz
Školitel	doc. Ing. Petr Tůma, Dr.
	petr.tuma@d3s.mff.cuni.cz
Katedra	Katedra distribuovaných a spolehlivých systémů
	Matematicko-fyzikální fakulta
	Univerzita Karlova v Praze
	Malostranské nám. 25, 118 00 Praha 1, ČR

Abstrakt

Zvyšující se popularita jazyků s automatickou správou paměti dělá z výkonnosti garbage collectorů (GC) klíčový prvek efektivního běhu aplikací. Bohužel, pro aplikační vývojáře není lehké porozumět chování GC z hlediska výkonnosti a návrháři výkonnostních modelů chování GC často ignorují.

V této práci (1) vyhodnotíme podstatu režie GC s ohledem na její vliv na přesnost modelů výkonnosti. Zhodnotíme možnost modelovat GC jako black-box model a zjistíme charakteristiky programů, které ovlivňují výkon GC. Poté (2) navrhneme analytický model jednogeneračního kolektoru a simulační modely jednogeneračního a dvougeneračního kolektoru. Tyto modely závisí na vlastnostech aplikací. Zhodnotíme přesnost těchto modelů a analyzujeme jejich citlivost na přesnost vstupů. Pomocí modelu ukážeme na rozdíly v chápání režie GC, pokud je založeno na znalosti algoritmu nebo skutečné implementaci kolektoru. Během vyhodnocování vlastností modelu objevíme z pohledu vývojáře aplikací důležité problémy s výkonem kolektorů a navrhneme jejich řešení. Nakonec (3) navrhneme model, který pomůže vývojářům předvídat jaké důsledky bude mít přidání určitého kódu do aplikace na celkovou režii GC. Tento model se snadno používá a požaduje pouze jednoduše dostupné vstupy a charakteristiky aplikace.

Klíčová slova

Garbage collection, výkonnost, modelování, Java

Contents

Chapter 1

Introduction

In the last decade, we read about a new programming language being introduced every few months. Whether a general-purpose or a domain-specific language, they tend to have one feature in common: automated memory management. The developers appreciate this kind of memory management because it simplifies their work by taking care of deallocating unneeded objects. It completely eliminates a whole class of errors known from languages with manual deallocations, namely errors caused by freeing an object too early, also known as *dangling pointer* errors. The second type of problem which the automated memory management can help to reduce occurs when the developer forgets to release some memory—leading to *memory leaks*. Here the automated memory management does not eliminate all possible leaks, but it has the ability to eliminate a large fraction of them.

The advantages however come at a cost. It is the need for an extra component included in the runtime system: the garbage collector (GC), responsible for reclamation of memory that the program cannot use anymore. It is only natural that adding an extra component to the system uses some extra computing power and therefore makes the system run slower. Today, typical garbage collector will be a generational tracing collector with copying young generation, which allows for very simple and fast allocation—definitely faster than classical malloc()-style allocations. Also, such collector will have higher throughput if we can give it more memory to operate with, making performance comparison between manual and automatic memory management tricky. According to Hertz and Berger [35] the overhead of the two approaches is on average equal if the automatic manager can operate in memory space five times larger than the space needed by manual manager. If the managed heap is only twice as large, the garbage collection will degrade the performance by 70 % on average.

Some developers avoid the environments with garbage collected heap because they believe their applications will be consuming too many resources. Others embrace garbage collection and value the increased productivity. Either way, managed languages are widely used, with Java being probably the most popular one. Large enterprise applications are often running J2EE technologies in data centers or in large clouds. Especially in large-scale setting, performance becomes an important factor—for example some cloud providers charge the customers based on utilization or number of active machines. It is therefore more and more important that the developers care for performance and understand the trade-offs they have to make. Garbage collection makes for an important part of the puzzle

when considering performance of the code the developer is working with. In this thesis, we want to investigate how the developer can understand what the effects of the code on garbage collection performance are.

1.1 Software Performance

When talking about performance of an application, or just a fragment of code, perhaps a single method, we can distinguish its asymptotic complexity and actual runtime, which we can measure when it runs on a computer. There is no doubt that an algorithm with the best complexity can be implemented in a way that causes the code to execute for an unacceptably long time. Traditionally, developers concentrate on asymptotic complexity—if we don't get the algorithm right then any code improvements, compiler optimizations, runtime speedup or even better hardware won't help. Looking from the other side, if the algorithm was right but the implementation itself was slow, the pragmatic approach often was to buy a better hardware or, in the case we already had the fastest hardware we could afford, wait half a year or a full year and then buy the hardware needed—there was a very good chance the improvements in hardware could compensate for sub-optimal coding.

Since around 2004, such remedy is generally not possible [71, 38]. Although processors are still more powerful, have more cores and larger caches, they are not faster in the terms of instructions executed on one core per unit of time. Around 2004, CPU manufacturers hit the physical barriers that prevented them from increasing frequencies as before, as illustrated in Figure 1.1, showing the frequencies of Intel CPUs over time. Now, the developers have to write code more cleverly—to use the more sophisticated and subtle features of todays processors, because the raw speed is not improving.

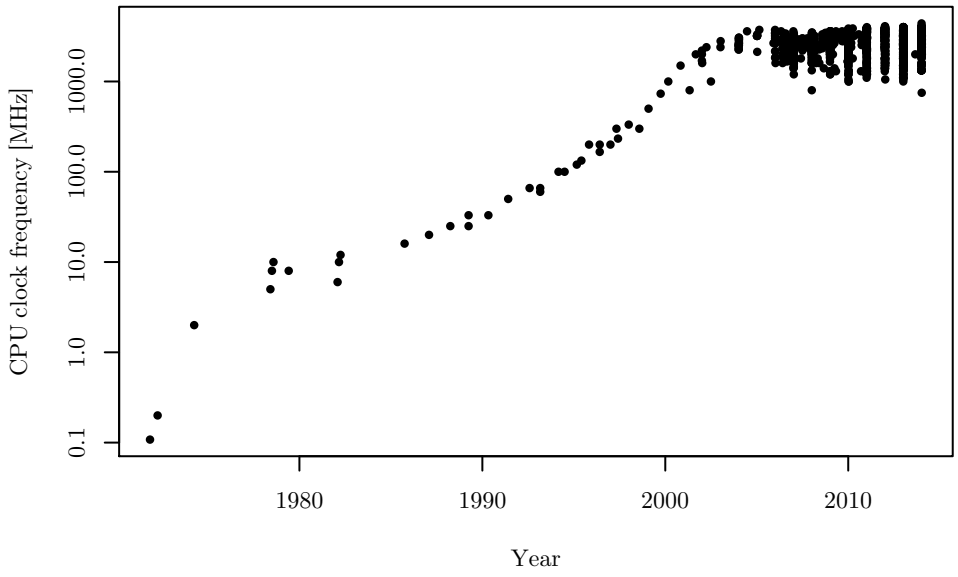


Figure 1.1: Intel CPU clock frequencies over time, data by [25]

The second example why today we have to care more about actual performance of software is the mobile world. Although mobile devices usually have

enough processing power, they have limited energy supply and the developers need to optimize their applications to use as little energy as possible. The third example we give is the cloud computing, where the bills for using infrastructure are typically based on utilization—unlike traditional model where the costs of owning and running computers are almost constant. In the cloud model, it may be worth the effort to optimize the code—if successful, it will translate into savings almost immediately.

Without a doubt, software performance is a broad topic attracting a lot of research. In context of this thesis, the most useful areas are performance modeling and evaluation.

1.2 Performance Modeling

One way how to improve the understanding of performance is to employ performance models of the software. Such models are simplified representations of the real system that have the capability to provide reasonable estimations of performance metrics we are interested in. We can typically differentiate models based on how detailed the underlying structure is.

On one end there are models with queuing networks or Markov chains as the mathematical foundation they rely on. Such models usually treat large portions of the system as a queue or a place and model results are based on request rates or similar high-level metrics. Often such models can have analytical evaluation which is usually very fast, for example simpler Queuing Network models [19].

On the other end we can find complex models based on architecture of the whole system or even reverse-engineered from running applications [45]. Such models will often employ model-to-model transformations to get from design or architectural model of the system to model suitable for quantifying performance metrics, such as Queuing Petri Nets [13] which combine queuing networks and Petri nets. These models are often too complex to evaluate analytically and they are evaluated using simulation resembling the execution of modeled application as described by the model. Notable representative of complex models is the Palladio Component Model (PCM) [14] which can be used to estimate performance of complex systems based on component development paradigm.

En example of a model that lies between aforementioned ends of spectrum is a model by Xu et al. [78] where they model Enterprise JavaBeans (EJB) applications using layered queuing networks (LQN).

PCM has a feature that allows the model developer to specify underlying system with another sub-model. For example if two components are connected by a middleware the designer can instruct the modeling framework to use a model of the middleware to evaluate also the effects of the communication in a more precise manner.

The modeling solutions mentioned above (PCM and LQN for EJB applications) have one property in common: they treat garbage collection as a constant background factor that the model can perhaps ignore or capture by calibration. In this thesis, we want to determine if this approach can impede the prediction quality of such models.

1.3 Performance Evaluation

We use performance evaluation to compare different systems, a system against expectations, or just to provide information on how a system behaves under some circumstances.

The advantage of experimental performance evaluation, when compared with modeling, is the fact that the application to evaluate is already available. If the measurement is done properly, it provides a definitive ground truth which is hard to argue with. Another question is the real meaning of the measurement results, because often the experimenter must handle many issues to provide repeatable and trustworthy results.

Every performance experiment needs a workload to run. If the goal is to evaluate performance of an application then it is the workload to experiment with. However, if we are about to investigate performance of a runtime system component, compiler or hardware, we need some code that will be using it. We can find standardized benchmarks like those developed by SPEC, targeted at different system components—CPU and compilers, JVM, graphics subsystem, MPI etc. Such benchmarks are believed to be representative enough to cover a spectrum of typical applications running on that system. This allows comparison between systems and gives the developers a basis for optimization of their products. For garbage collection in JVM, the widely used benchmark suite is DaCapo [17].

A specific connection between performance evaluation, benchmarking and modeling are black-box models. They are built by first identifying the workload parameters or inputs that are important for the performance of modeled system. Then a series of measurements establishes what the performance metric of interest is for subset of possible input and parameter combinations. The model is then created as a (multi-dimensional) function of parameters and inputs. Values from the combinations that were not measured have to be interpolated or estimated otherwise [33, 24].

When evaluating the code running in managed environment, the community is not unified on how to handle such situations. Three basic approaches are common. First is to force garbage collections between benchmark iterations, causing each iteration to start with the same heap state. Second approach is to set the collector and heap parameters so that no collections occur inside the benchmark iteration, usually combined with forcing the collections between iterations. Third approach is to take no special actions regarding garbage collection and keep the default behavior. All the three approaches have their advantages and disadvantages and it is difficult to choose the correct one for the task at hand.

Another issue comes from the applicability of the results. If we evaluate the workload in an isolation, it is not clear how relevant the observed collector performance will be when the code executes in a larger environment. A similar problem may arise with accounting of the machine utilization. In case the code executes in short intervals it will probably not trigger the collection, making someone else to pay for the cost of memory allocations.

Due to the tracing garbage collection nature, the more space it has available the more effective it is, the users have to manage heap settings, analyzing the speed-versus-space trade-offs. The typical questions asked are whether it is better to fix the parameters by setting them manually or rely on the automatic sizing.

1.4 Research Goals

Guided by the open issues summarized in previous sections we set the following research goals:

Goal 1: *Examine the nature of collector overhead.*

We want to learn what is the nature of garbage collection overhead with respect to performance modeling. We want to find out if the collection overhead can be modeled as a composition of relatively simple factors to facilitate black-box modeling. We also want to determine if it is reasonable to model the overhead as a constant factor and if not, then if the overhead is significant enough to require including into the modeling infrastructure.

Goal 2: *Understanding the contribution of algorithm and implementation.*

Collector performance is determined both by the general algorithm employed and by the implementation used. In this context, the goal is to determine if understanding the collector based on knowledge of the underlying algorithm is enough to understand the performance behavior on a specific workload, or if we need to understand also the details of particular collector implementation to explain how the collector performs. We also want to identify those information details the developers should know to achieve more predictable performance.

Goal 3: *Insight with reasonable information.*

We want to identify situations in the development process where we can provide support for developer's decisions. While first two goals are from the perspective of model designers, this goal aims on application developers. We want to provide support in situation when the developer needs to make a decision that will affect performance of the end product but he does not have the code to measure yet. We want also such support to be practical—without excessive data collection or time needed.

If we could find an accurate and simple performance model of garbage collection that is based on characteristics of application behavior, it would fulfill all the goals. However as we investigate the topic we find that such model would need to be as complex as the collector itself and we have to find different means to tackle our goals.

1.5 Structure of the Thesis

We start with a brief overview of garbage collection algorithms and one garbage collector which we use in most of our experiments, complemented with related work overview in Chapter 2. In the next three chapters we delve into investigation of the presented goals.

We start with investigation of collector behavior in terms of black-box modeling. We define workload characteristics that should be important to the collector performance and design workloads to exercise the collector in those aspects—we use artificial workloads and their combinations. With these workloads we run extensive experiments to quantify collector behavior. The experiments reveal that

the behavior of the collector is unlikely to be modeled as a black-box—instead we need to concentrate on a selected collector whose implementation we investigate thoroughly. Some of the measurements will also show the fine-grained performance models should try to include garbage collection into them. We published this investigation in [51, 48, 4] and we present it in Chapter 3.

Our next step is the definition of still simplified, but already very detailed model of a common garbage collector (parallel throughput collector in HotSpot VM, described in Section 2.4). We investigate what precision can be expected from a model based on the application behavior regarding object allocations and reference manipulations—inputs consist of information about all allocated objects and reference updates. We evaluate the accuracy of the model using a simulator on several benchmarks. We describe some issues that reduce accuracy of such a model and analyze the impact of the reduced input on accuracy. At the same time we discover a problem with data collection present in current lifetime profiling tools. We show that the collector models cannot be expected to have a good accuracy even with extremely large inputs. We also show some examples of very counterintuitive behavior. We present this investigation in Chapter 4, published in [49].

After showing that detailed simulation needs impractically large amount of data and still achieves only mediocre accuracy, we attempt to find a situation in which we can provide an insight with easily obtainable input. We concentrate on a scenario where a developer plans to add a feature into existing code. We investigate what are the properties of the original code and the new feature, that allow us to define a model to predict the effects of the new feature on garbage collection performance. We find that we can provide reasonable predictions if the original code is in steady state and has stable allocation behavior and the new feature allocates short-lived objects only. We define and evaluate such model in Chapter 5, published in [50].

We conclude the thesis in Chapter 6.

1.6 Notes on Conventions

[X]

The text and plots of the following three chapters are partially based on publications mentioned in Section 1.5. The parts we included from those publications verbatim are marked with a vertical line on the inner margin. Within one chapter all marked text comes from publications mentioned at the beginning of the chapter and also at the start of the line. This text is marked as if it was originally published in [X].

If the line is black we made no modifications apart from terminology unification etc. If, as on the margin of this paragraph, the line is red, we made signifficant modifications that are beyond the original publication.

In the literature related to garbage collection the *GC* acronym is used for both *garbage collection* and *garbage collector*. We also use the acronym for both concepts. We believe the attentive reader will understand which one we mean from the context, if distinction is necessary.

Chapter 2

Garbage Collectors

In this chapter we describe garbage collection principles and some garbage collectors, with special attention on the *throughput* collector from Oracle's HotSpot JVM which is our collector of choice in the following chapters. This will not be a complete survey, the interested reader can rely on very good book *The Garbage Collection Handbook: The Art of Automatic Memory Management* by Jones et al. [40] providing comprehensive compilation of GC principles, algorithms and methods.

We start with explanation of basic garbage collection terminology, goals and principles in Section 2.1. Then we present four fundamental collection algorithms in Section 2.2: reference counting, mark-sweep, mark-compact and copying collection. In Section 2.3 we discuss the generational garbage collection, followed by the description of HotSpot throughput collector in Section 2.3 and short overview of other collectors in Section 2.5.

We do not cover at all the large body of work related to garbage collection related to real-time systems. The principles and goals are usually different then for classical non-real-time environments. Even such specialties like collector implementation in hardware were developed for real-time systems [10].

2.1 Principles and Terminology

Garbage collection was introduced by John McCarthy as a part of the LISP system in the 1960 paper [55]. It was rather short (one page) presentation of what we would call a mark-sweep collector today. Since then, the garbage collection development and design have advanced considerably but the basic goal is still the same: return the memory location that is allocated by the program, but it cannot be used by that program anymore, back to the environment to facilitate consecutive allocations.

In this work we use the application model and terminology similar to that used currently by the garbage collection research community. The application executes the code within the *virtual machine*, storing local variables and function activation records on the call stack. It uses *allocator* to get access to additional data structures in form of *objects* on the *heap*, using allocate operations. The heap is continuous or comprises several smaller continuous *spaces*. The objects consist of several *fields*, of which some are *references* pointing to other objects or

pointing nowhere—null references. The virtual machine implementation usually adds a *header* to the beginning of every object and may enforce *alignment* on placement of objects in memory. The application can also have global variables and *static* fields, which act the same way as the global variables do, and some of them can be references.

The application consists of one or more *threads*, with each having its own call stack. From the perspective of garbage collection, the program consists of two parts: the *mutator* and the *collector*. The mutator executes the "real" code of interest, from the GC perspective it can execute allocate operations, a read(object, field) operation to access the value stored in some field of a specified object and a write(object, field, value) operation to modify some field of a specified object to the given value. If the collector needs some instrumentation in the write (in read it is very rare) operation, it provides the mutator with a piece of code called *write barrier* and the mutator is obliged to execute it with every such operation.

We restrict ourselves to type-safe languages, which cannot construct a reference arbitrarily or convert some other value—like an integer—to a reference (Java, Python, ...). The non-null fields and variables of reference type that the mutator has direct access to are called *mutator roots*, or *collection roots* if discussed from the perspective of the collector. Typically, roots include all local variables from all threads, static fields, global variables and references stored in registers. If the application needs to access some object in the heap it has to follow a reference chain starting at one of the roots. The object is *reachable* if there exists such a chain, starting at any root. Otherwise, the object is *unreachable*.

It is safe to *reclaim* (deallocate, free) the memory of unreachable objects, because the application has no means to access it again. The reclamation of unreachable objects is the job of the collector, leaving reachable objects in the heap. The set of reachable objects is usually bigger than the set of objects that have to stay in memory—only objects that will be accessed again have to be kept there. These objects are called *live*. Objects that will not be accessed until the program termination are called *dead*. Ideally, the collector would reclaim all dead objects. However, the liveness property is undecidable and the collectors have to refrain to weaker property—reachability. This is the reason why even in garbage collected heaps memory leaks are possible. Since we cannot really reason about liveness as it is, we will further use live and reachable (dead and unreachable) as synonyms. We often use the term *garbage* for unreachable objects as well. If the collection reclaims all garbage, it is called *complete*.

Collectors operate either in *stop-the-world* manner, meaning the operation of the mutator is paused when the collector operates, or *concurrently* with the mutator, in one or more threads running in parallel or interleaved with the mutator. Some collectors combine the two approaches and have both stop-the-world phase and concurrent phase.

2.2 Basic Collection Algorithms

We distinguish two fundamental garbage collection algorithm classes: *direct* and *indirect*. Direct algorithms operate with object reference creation and deletion. The representative algorithm is *reference counting*, which we present first. The

indirect class algorithms are walking through the reference graph and establish which objects are reachable. Then they (indirectly) deduce that everything they did not see must be garbage. Because they trace all references from the roots, these algorithms are also called *tracing* garbage collectors. Three fundamental tracing collector types are mark-sweep, mark-compact and copying collection and we present them briefly after reference counting.

2.2.1 Reference Counting

Reference counting was first published by Collins [23] just several months after McCarthy's paper introducing the tracing garbage collection. The basic idea is straightforward—every object maintains a counter of how many references are pointing to it. When the count drops to zero, the object can be deallocated. In the deallocation process, it effectively deletes all references pointing from the deallocated object and therefore decrementing reference count of target objects, possibly generating a deallocation cascade.

A simple implementation uses a counter in the object header that is large enough to hold the biggest possible reference count and a write barrier intercepting all reference updates. This includes updates in registers or local variables. Such implementation exposes two major problems: it can't handle reference cycles and it has a big overhead. In reference cycles, the counters will never decrease to zero and therefore the objects will not be identified as garbage even if there is no external reference pointing to the cycle. For the overhead problem, imagine we want to iterate over a long linked list, keeping the reference to currently visited node in a local variable. Advancing to the next node then includes decrementing the counter in the current node and incrementing the counter in the next node. When we add to this the need for synchronization in parallel environments, the resulting performance will be poor.

To tackle the problem with reference cycles, one of two approaches is usually adopted: using backup tracing collector or trial deletion. The first method [76] ignores the cycles in the reference counting implementation. Then, if the program is using some data structures with cyclic references, it will run out of memory eventually and at this moment it can execute a tracing collection to identify the garbage.

The trial deletion method finds a candidate that could be a part of an unreachable reference cycle and simulates what happens if that object is deleted, effectively removing cyclic references including the candidate object. In successful cases the objects are identified as garbage and can be collected. The trial deletion was introduced by Christopher [22], later it was optimized for example in [53, 11].

To reduce the overhead of the reference count update, two independent approaches are being widely used: *deferred* and *coalescing* reference counting. In deferred counting [27] the collector does not increase or decrease reference counters on updates in registers or local variables, only changes in references from heap object fields are being processed. When the counter drops to zero, it cannot be reclaimed, because it can be referenced from the registers or local variables—generally from the mutator roots. Therefore such objects are recorded in *zero count table* and in a later phase (the garbage collection) the roots are scanned

and if none of them is pointing to an object from the zero count table and the object has still a count of zero, it can be reclaimed.

The coalescing reference counting [47] exploits the observation that if we look at the reference at the beginning and at the end of some interval, we only need to update the counters reflecting the change at the edges of the interval. If, at the beginning, the reference points to object A, then during the interval it points to objects B and C and then points to D, which remains referenced at the end of the interval, we can act as if the reference changed directly from A to D, because the effect on objects B and C will not persist outside the considered interval. This idea can be used to reduce counter updates but introduces garbage collection phases (as also deferred counting does).

A study evaluating performance of fully optimized reference counting against tracing collector was published by Shahriyar et al. [66].

2.2.2 Mark-Sweep

The algorithm consists of two phases: first the reference graph is traversed, starting from roots (registers, thread stacks, global variables, static fields) and the objects visited on the way are *marked*, therefore it is called the *mark phase*. The *sweep phase* then traverses all objects in the heap, leaving those marked intact and reclaiming everything else.

The runtime environment must possess the ability to enumerate the roots, which may be sometimes difficult and require deep runtime and collector integration. Moreover, if it is possible to safely identify reference fields in the heap objects, the collection will be complete. Some *conservative* collectors for languages like C/C++ can't identify pointers reliably and assume that everything that looks like a pointer is a pointer. The collection is then incomplete. Simple implementation can execute the marking traversal with a stack as the work list, which results in a depth-first traversal order.

The theoretical description of marking is using three colors, or states, of the objects during the collection—the *tricolor* abstraction [29]. The object can be *black* or *white* at the end of marking, denoting a live object or garbage. All objects are white at the collection start, when the marking discovers a new object it is marked *gray*, and when it is fully processed (outgoing reference targets included in the work list) it turns black. The algorithm ends when there are no gray objects. The practical meaning is that black objects are live and fully processed, collector knows about gray objects but needs to process them (work list) and white objects are possible garbage. This provides a nice invariant: after every marking iteration, there are no references from black to white nodes, which can be used to prove correctness. It can also be used for concurrent marking, when the objects allocated while the collector is traversing the heap are allocated as gray.

The implementation needs to distinguish only two colors for every object—black and white—commonly implemented as one bit in the object header. Other option is using a mark bitmap, it is necessary especially for conservative collection.

Over time improvements to performance were investigated, for example marking with a stack as a work list can thrash the caches heavily. Cher et al. [21] proposed a solution where the objects picked from the work list are not processed

directly, but they are stored in a FIFO queue of a specified length. This queue gives a chance for prefetching and can improve performance. Another optimization strategy is lazy sweeping [39], where the sweep is not executed all at once but only in small chunks during the allocate operation to decrease the pause times.

2.2.3 Mark-Compact

One of the bigger mark-sweep problems is heap fragmentation. It can be improved using smart allocation strategies but this can give no guarantees. Compacting solves the problem by putting all live objects together and leaving the free space in one piece, making the best use of memory and allowing for very fast *bump-the-pointer* allocations. Mark-compact collectors are using the same mark phase as mark-sweep collectors, with some algorithms requiring to use mark bitmaps. Because compaction moves objects, it has to be careful about references to the moved objects. It can use indirection and fix only the reference handle values, which is simple for the collector but adds extra overhead to the mutator, or it has to update all references pointing to moved objects. Here we will present two compaction algorithms: the Lisp 2 algorithm and the one-pass algorithm.

A Lisp 2-type collector is using three passes over the heap and needs every object to have a *forwarding address* field. It performs so-called *sliding compaction*, where all the surviving objects are moved to the beginning of the heap in the same order they were in before the compaction. The first pass goes over all marked objects from the start of the heap and computes the address where it will copy the objects—the first live object to the start of the heap, the second one right after the first one and so on. The computed address is stored to the forwarding address field. In the second pass, the objects are still in their original locations, and the collector updates references. It traverses all references in the heap, looks at the forwarding address field of the reference target and updates the reference to the new value. The third pass copies the object to their destination location, being careful if the original object location and the new one overlap.

The Lisp 2-type compaction is simple but it has the disadvantage of three heap traversals. If we want to reduce the number of passes, we need to calculate and store the forwarding addresses in some helper data structure. High performance algorithms with just one pass over the heap [1, 44] rely on a special version of the marking bitmap. During marking, the collector sets the bits corresponding to the beginning and the end of the object, so it is possible to compute object sizes just from the bitmap. The heap is logically partitioned into small blocks of the same size (*regions*) and the collector can calculate in one pass over the bitmap where the first object from the block will be copied. This address is stored in the offset vector, which is an additional data structure the collector needs. Then, in one pass over the heap, live objects can be copied to the start address of the block plus the sizes of the preceding objects in the block. When an object is copied, the collector also updates the references—new values are computed using the original address: the new address will be the address in the offset vector of the corresponding block plus the size of objects in the block, which can be calculated from the bitmap.

When comparing the one-pass algorithm with the one of Lisp 2 type, the first will make less memory reads, because it will traverse the heap data only once

while the Lisp 2 algorithms traverses the whole heap three times. However, the one-pass algorithm needs extra data structure (the offset vector) what makes the choice between the two types interesting.

2.2.4 Copying

The third fundamental tracing collection type is copying, here we will present its basic form, the *semispace* copying [30, 20]. The algorithm divides the heap into two spaces of the same size, *fromspace* and *tospace*. New objects are allocated in tospace using the bump-the-pointer technique. When there is no space left in the tospace, the collection starts. First it flips the roles of fromspace and tospace, so all the objects are now in fromspace. The collector traverses the reference graph from the root and it copies (*evacuates* or *scavenges*) the objects it visits into tospace. After collection, the fromspace is not needed.

The collector moves objects, so it needs to fix references. In case of copying collector, this can be achieved using forwarding pointers in the fromspace copies of the copied objects. Because the fromspace will not be used by the mutator anymore, the collector can use any field of the object, not just a field dedicated for this purpose. When the object is copied to tospace, it is conceptually gray, and the forwarding pointer in the old fromspace copy points to the new location in tospace. When the collector processes this object, it scans its references and if their target (which is now pointing to fromspace) is an object not yet evacuated, it copies it to the tospace and updates the reference value. Already evacuated objects have their forwarding pointers set, the collector just updates the reference to that of the forwarding pointer. Finally, the object is turned black.

Because the objects are copied one after another as they are visited, the collector does not need to use an extra work list (stack)—it can use directly the objects already copied into tospace that are not yet fully processed (gray color). For that, it only needs one extra pointer.

When comparing copying collection with mark-compact, copying is faster, allows equally fast allocation, is easier to implement, but it uses twice as much memory.

2.3 Generational Collection

Tracing garbage collectors, especially copying, are most effective on heaps with few live objects. In that case it only traverses those few live objects and releases large amounts of memory. However, most applications have some objects that survive many collections and the collector has to trace them (and copy in case of copying and compaction) many times in what is a heavy work with little effect. Therefore many schemes have been introduced that try to divide the heap into several independently collected regions and use the collector that has the best performance in that type of region. Over time, probably the most successful scheme is generation collection, where the objects are divided into regions based on their *age*.

This approach relies on the *weak generational hypothesis*, which postulates that most objects die young [74]. It appears to be valid in most applications (i.e. [28, 17, 41]). On the other hand, the *strong generation hypothesis*, that of all

objects (not only newly allocated), the older objects are expected to have higher survival rate, does not have so widespread validity [34].

From this comes the assumption (of course, there can always be an application where most objects die old) that it is useful to divide the objects by age into two or more *generations*. Each generation can be collected separately and with a different algorithm, however, usually collecting an older generation means also collecting all younger generations. Objects are allocated into the youngest generation, called *nursery* or *eden*.

The advantage is that objects newly allocated in the nursery, which is properly sized, usually also die there. For example, according to Blackburn et al. [17], in SPECjvm98 benchmark suite, on average only less than 9 % of object survive longer than 4 MB of other allocations. It means collecting this space will be relatively cheap.

Old enough objects can be *promoted* or *tenured* to an older generation, where the objects that tend to live longer are located. However, many objects do not reach this older generation and therefore collections are less frequent, which should pay for the higher collection cost.

For possibly better collection performance, the system as a whole needs to sacrifice some mutator performance. The problem lies with collecting generations independently: the object in a generation is reachable if a reference trace exists from the roots, using all objects in the heap, not only objects in the same generation. Without extra information, the collector would need to do the tracing of the whole heap, ending up with very little performance benefit, if any. The generational collectors therefore record inter-generational references and add them to the set of collection roots. The mutator executes a write barrier on every reference write and checks if the reference is inter-generational. If it is, the source object is recorded into a *remembered set*. At collection time, the objects in the set are scanned for inter-generational references still pointing into the collected generation and added to the roots.

This is also one of the reasons why a higher-generation collection usually also collects all younger generations—in that case there is no need to record references from younger to older generations, only from older to younger.

Typically, generational collectors are using two generations. The *young* generation is either all dedicated to nursery, where the objects are allocated, or it can consist of more spaces, but all of them are always collected together. Then there is the *old* or *tenured* generation. A collection of the young generation is often called *minor*, the collection of the whole heap is *major* or *full*.

2.4 HotSpot Throughput Collector

In this section we will describe more precisely our collector of choice (we explain the choice later) for the most of the experiments in the following chapters—the *throughput* collector from Oracle's HotSpot JVM, where it is the default collector since version 1.5 (or it can be enabled using `-XX:+UseParallelOldGC`) startup option. The information in this section is based on available literature ([70, 69, 60, 63]), mailing lists, other online sources and study of the OpenJDK source code.

The collector is generational, with two generations and stop-the-world collections. It has four spaces for normal heap objects and one extra space for virtual machine objects (permanent "generation" or metaspace), with layout similar to the original Ungar's Generational Scavenging [74]. The young generation consists of three spaces: one eden and two survivor spaces, the fourth space is for the old generation.

New objects are allocated in the young generation, specifically into the eden space, using the bump-the-pointer technique. When the eden is full, the young generation garbage collection is triggered (minor collection). It is using a copying algorithm, with live objects being scavenged into one of the survivor space, along with the objects from the second survivor spaces. At the end of young collection, eden and the second survivor are left empty, allowing fast bump-the-pointer allocations. The roles of the survivors are flipped, in action similar to semispace copying collection.

The collector defines the age of young generation objects as the number of young collections survived. It is recorded in the object header, using 4 bits (older versions were using 5 bits), allowing for a maximum age of 15. The collector defines a *tenuring threshold*, objects older than this number are promoted to the old generation during young collection. If there is not enough space in the survivor, the objects are copied directly to the old generation, causing *premature promotions*.

This is a generational collector, so for correct young collection operation it needs to know about pointers from the old generation to the young. The remembered set is implemented using a *card table*, where for every 512 bytes of the heap (one *card*) there is 1 byte in the card table. The mutator is using a simple write barrier that marks the card as dirty in the card table whenever a reference is written into a field in the corresponding part of the heap. One could possibly use just one bit, but then difficulties arise in multi-threaded environments when setting a single bit is not an atomic operation, while setting the whole byte has no danger of losing update. At the beginning of a young collection, the objects in the dirty cards of the old generation are scanned for any references pointing into the young generation and these are added to the collection roots.

To permit fast allocation in applications with more threads, *thread-local allocation buffers* (*TLAB*s) are used. Each mutator thread allocates its own large contiguous piece of heap—the TLAB, from which subsequent allocations are served using bump-the-pointer technique. When the thread fills its TLAB, it allocates the next one, which should be an infrequent operation and the cost of needed synchronization is expected to be low. Large objects that do not fit in TLAB can bypass the local buffer and be allocated directly into the eden, or directly into the old generation if they don't fit in the eden. Similar optimization is used to facilitate promotion into the old generation or copying into the survivor space using more threads. The collector threads allocate *PLAB*s (promotion local allocation buffer) in the target space and then use bump-the-pointer allocation without need for synchronization.

The young generation collection can use more than one thread (it is called *Parallel Scavenger*). It can happen that more threads discover and attempt to copy an object at the same time, therefore the collector has to handle such races. When the collector thread discovers another reachable object, it tries to copy

it into the survivor space or the tenured space, depending on its age. First, it allocates space for the object, handling any out-of-memory conditions. Second, it copies the contents of the object to the target space. Third, it attempts to install a forwarding pointer in the original object header. This is done atomically, using compare and swap technique where only one thread will succeed. The winning thread will continue processing the object, other threads have to undo the allocations.

Objects promoted from the young generation into old will fill the old space eventually. When the old generation is almost full, a *full* collection is triggered. The almost-full condition means the collector calculates a weighted mean of the amount of data promoted into the tenured space and also the weighted deviations from that mean. The collection is triggered when the free space is less than the weighted mean increased by three times (by default) the weighted deviation. The full collection operates on the whole heap using a mark-compact algorithm similar to the one described in Section 2.2.3 as the one-pass algorithm. It has three phases: mark, summary and compacting with updating references. The first and the third phase can be executed using more GC threads, the summary phase is always single-threaded.

For the compaction purposes, the heap is divided into multiple regions of equal size: 1 KB on 32 bit systems and 2 KB on 64 bit systems. The marking phase is executed using multiple collector threads, which mark all live objects. It is using mark bitmap for the marks, while handling concurrent updates. Moreover, it calculates the size of live data in each region while marking, the additions are atomic.

Each thread has its own worklist. It picks the object from the worklist, processes it and puts unmarked children to the worklist for further processing. If its worklist is empty, it steals the work from some other thread's worklist, which is implemented using deques for the full collector. The young generation collector is also using work stealing for load balancing.

The summary phase, always single-threaded, calculates the *dense prefix* and pre-calculates where the live objects will be compacted later. The dense prefix is the start of the compacted heap where most objects are live—immortal and long-lived objects will eventually end up there and it is not necessary to compact this area. The prefix calculations operate on regions. From the space start, all regions full of live data are considered dense, then the algorithm searches for the first region that has approximately the same live/dead ratio as the whole old generation. The actual dense prefix is set in between the first region with dead objects and this limit region. As the first region to compact will be selected the region that will cause the compaction to achieve the best data reclamation ratio. For the rest of the regions it computes how much live data (the dense prefix is considered all live for this purpose) is before each region, allowing to calculate the start address where the data will be compacted. The summary also calculates where the first live object is in the region or how much live data extends to the region from the previous one.

The next phase copies the live objects and updates references. Regions that are empty or will be compacted only to themselves are available as target regions. The compaction threads pick work from those regions and copy objects from their

respective source regions while updating references. It is guaranteed that there will always be at least one such region. Available regions are maintained in an atomically managed list. The objects that are on the region boundary are also copied but the references from them are updated later.

During full collections, the permanent generation or metaspace is also collected. The permanent generation is collected with a serial mark-compact collector. Since JDK 8, the permanent generation was discontinued and the JVM is using metaspace for class metadata. There, the objects are allocated in native memory and can be deallocated when the classloaders get collected. Both mechanisms can induce a full garbage collection when they run out of space, hoping to collect some classloader to be able to free some space.

This is a stop-the-world collector—all mutator threads are stopped during garbage collections. However, the collector needs to know the threads are paused in a safe state to allow correct collection—for example a moving collection can move objects that are referenced from the root set, for example from the registers and the collector has to update those registers. For that, the mutator thread code is instrumented with *safepoints* in well chosen places where it is safe to do a garbage collection, and the virtual machine has a roots map for all safepoints so it knows what the thread roots are.

The collector employs a set of ad-hoc rules called *ergonomics* that can size the heap spaces so that the collector pause times, throughput and footprint do not exceed goals that the user can set. The ergonomics can be turned off by `-XX:UsePSAdaptiveSizingPolicy` command line option. The strategy is first to meet pause time goal, and only if it is met the collector targets the throughput goal and if also that one is met, the attempt to meet the footprint goal is made. Improving the pause time goal can be generally achieved by making the young generation smaller (there is nothing that can be done with full collection pause time, it depends on the number of live objects in the application). However, if the throughput goal is not met the collector attempts to increase the heap size to make the collections less frequent, which can again break the pause time goal. The footprint goal decreases the heap size until the throughput goal is not met (and then the throughput goal priority causes increasing the heap size).

2.5 Other Collectors

The design space where garbage collector designers operate is vast, therefore there are literally hundreds of different garbage collectors available. We only mention some of them here, the interested reader can surely find more details when needed, good source to start with is The Garbage Collection Handbook [40].

In HotSpot virtual machine two more principally different collectors are available: the Concurrent Marks-Sweep (CMS) collector that can run in parallel with the mutator or in the *incremental* mode where the collector executes only in short pauses that suffice to proceed only in small steps. Then there is the G1 collector [26] that divides the heap into many small pieces and always tries to collect the piece with least live data.

The source of many garbage collectors is the Jikes RVM [3] with its MMTk framework [16], for example collectors like Immix [18] are implemented there.

The GC research is active also in other areas, like functional languages [73] or highly parallel platforms [12].

There is no single best garbage collector for all workloads and requirements. As shown in a study by Fitzgerald and Tarditi [31], where they used 20 benchmarks with 6 modern collectors, for every collector they could find at least one benchmark that runs at least 15 % faster with a more suitable collector.

From here also stems our choice for the primary collector we use in this thesis—the parallel throughput collector from HotSpot VM. We know it is not the best collector (there is no such thing as the best collector), but it is a part of probably the most popular Java virtual machine available, it is the default collector there and it is widely used. It is also a generational collector which makes it more interesting than one-generation collectors, but on the other hand it is one of simpler ones because of its stop-the-world nature.

2.6 More Related Work

In addition to publications mentioned before in this chapter, a large body of work related to this thesis comes from the garbage collection design and evaluation community. Researchers in this community design complete collectors, optimizations of more or less important parts of collectors, or evaluate which collector is best for a specific task. Nowadays work on real-time garbage collectors is also frequent. A comprehensive database of publications related to garbage collection is maintained by Richard Jones, the coauthor of The Garbage Collection Handbook [40]. It is available at http://www.cs.kent.ac.uk/people/staff/rej/gcbib/ and contains more than 2500 entries at the moment. We also provided several references on the topic in previous sections of this chapter.

Related work on performance modeling of garbage collection is scarce, we are aware of only two relevant papers. First paper is authored by Vengerov [75], who derived an analytic model that can be used to adaptively size generations. Unlike our work, his model uses metrics such as the amount promoted in a young collection for the input and therefore it is not very helpful for an application developer. The second model is by White et al. [77], who use a PID controller to adapt the heap size while measuring the collector overhead. This model is again based on observation of internal GC metrics and can't provide insight for the application developer.

An interesting work on distinguishing effective and excessive memory usage by Mitchell and Sevitsky [56] defines a health signature enabling distinction between bloat and healthy memory usage. They allow the developer to decide if his application is using memory in an effective manner—if not, this has a definite effect on GC performance.

A large portion of this thesis is based on experimental evaluation, which forms another body of related work. It consist of benchmark construction [17, 68], measurement and evaluation methodology [67, 32, 43, 37] or analysis of observed results [57, 58], to name just a few.

Throughout the thesis we rely on more publications. These are usually specific to the investigation at hand and we therefore cite the sources there.

Chapter 3

Empirically Investigating GC Overhead

Today, according to various lists of programming languages popularity (i.e. [72, 59]), more than half of the Top 10 most popular languages are those with automated memory management and therefore also with garbage collectors. In such settings, performance of the collectors is of much importance and it is only natural that the research community evaluates the overhead of their algorithms very thoroughly [26, 79, 17].

On the other hand, the performance modeling community tends to treat the garbage collector only as a constant background factor that can be captured implicitly in calibrated performance models [78, 14, 46]. When garbage collection is to be considered, one option for the model authors is to study the internals of the platform and adjust the model accordingly. However, for example in the Oracle HotSpot virtual machine, the collector implementation has more than 50000 lines of code. It is unrealistic to expect the model authors to be able to study such a large body of code for each considered platform. Another method may be based on documentation, which describes the algorithm in reasonable detail [70], but even such documentation can still make unrealistic claims, such as the collector keeps its overhead below 1 % of total execution time [69]. As shown by Blackburn et al. [15], the collector can easily make application three times slower.

In this chapter, we measure the garbage collection performance using artificial workloads to find out if and when the treatment of garbage collection as a constant background factor is reasonable enough and if we can find basic characteristic of the collectors that could be used as a basis for potential collector models. We apply a methodology that proved to be useful with other resource sharing models—devise artificial workloads that stress the resource (collected heap) in a particular way and thus determine a partial function of performance depending on a workload characteristic [7, 8, 5, 9, 6].. In a sense this is approach inspired by black-box modeling as investigated by Happe et al. [33] to model messaging middleware.

The chapter is based on these two papers:
[48] P. Libič, P. Tůma, and L. Bulej. Issues in performance modeling of applications with garbage collection. In *Proceedings of the 1st International Workshop on Quality of Service-oriented Software Systems*, QUASOSS '09, pages 3–10, New York, NY, USA, 2009. ACM
[51] P. Libič and P. Tůma. Java garbage collector performance measurements. In J. Šafránková and J. Pavlů, editors, *WDS'09 Proceedings of Contributed Papers: Part I – Mathematics and Computer Sciences*, pages 34–40. MATFYZPRESS, June 2009 [48, 51]

In this chapter, we investigate the issues related to GC overhead from the performance modeling perspective, looking at some of the more pressing questions like:

- is the overhead significant enough to warrant explicit attention in performance modeling?
- is the overhead a constant factor that can be captured implicitly in calibrated performance model parameters?
- does the overhead depend on reasonably well defined external factors that could be potentially included in performance models?

The experiments used to answer some of the questions are presented in Section 3.1, with some implications of the results in Section 3.2.

3.1 Collector Performance

The GC overhead obviously depends on the particular GC implementation. Results of any performance evaluation experiments are therefore immediately relevant to that implementation only. While such results are useful in studies of proposed GC algorithms, they are less useful in constructing performance models—such models would be tied to particular service platform internals and many changes that are usually considered trivial from deployment perspective (e.g. backwards compatible virtual machine upgrade) would have the potential of making the models irrelevant.

The option of designing performance evaluation experiments based on detailed knowledge of a particular GC implementation—tempting as it is from the engineering perspective—is therefore less beneficial in the context of service performance modeling. It would produce too complex models with too limited applicability as we shall see in the rest of the thesis. Instead, we attempt to design experiments that are likely to exercise any GC implementation, focusing on general factors such as heap size, allocation speed, object instance lifetime or object graph depth. While the results are still immediately relevant only to a particular GC implementation, and therefore do not allow us to make generalizing claims, they still provide practically useful information for that implementation. We also compare some of the experiment results for two different collectors to see if generalizations are possible.

3.1.1 Workloads

All the experiments described later are based on three basic workloads or their combinations. The workloads exercise different aspects of the GC implementation depending on multiple parameters. Besides the configurable parameters associated with the workloads, the GC implementation provides additional tunables such as maximum heap size. An experiment then defines an instance of a particular workload type, with some of the parameters varied during the experiment and others kept fixed. To allow result comparison between different workload instances, the implementation of the workloads has been carefully designed so that changing the workload parameters does not change the exercised code paths (in terms of major branching directions, major iteration counts, etc.).

3.1.1.1 Object Lifetime

The GC implementations in production service platforms are mostly generational and therefore can treat each generation differently. The object lifetime workload was designed for experiments quantifying dependencies in GC behavior with respect to lifetime of heap objects. During execution, the amount of memory consumed by live objects is constant, garbage is generated at constant rate, and the heap shape does not change.

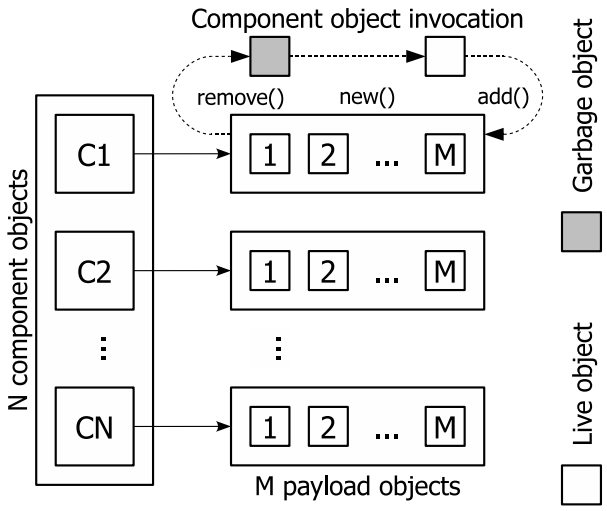


Figure 3.1: Object Lifetime workload

The basic object structure of the workload is depicted in Figure 3.1. The workload uses a predefined number of *component objects*, meant to represent relatively stable application objects. Each component object contains a predefined number of *payload objects*, meant to represent relatively temporary application objects. The size of the payload objects, as well as the numbers of component and payload objects can be set for each instance of the workload, and remain constant during workload execution.

The workload repeatedly invokes a single operation on the component objects. Upon invocation, a component replaces the reference to its oldest payload object by a reference to a newly allocated payload. It then performs certain amount of phony work using the payload objects to regulate the object allocation speed.

When the operation completes, it is immediately invoked again, either on the same component or on the next component in the list, depending on the desired effect of the workload.

The effect of the workload is determined by the sequence of component object invocations. When we desire the heap to only contain young objects, the workload loops through all the component objects and invokes the operation once on each component object. When we desire the heap to contain mostly old and some young objects, the base loop is the same, i.e. the operation is invoked once on each component object, but before advancing to the next object, the operation is always invoked a predefined number of times on the first component object. This makes the payload objects of the first component young, because they are replaced frequently in comparison with payload objects from the other components. The heap therefore contains two classes of payload objects with different lifetimes, with the ratio between the allocation speeds equal to the inverse of the number of consecutive invocations performed on the first component object.

3.1.1.2 Heap Depth

The heap depth workload was designed for experiments quantifying the dependencies of GC behavior on the depth of the heap, measured as the number of steps needed to reach the leaf objects from the root references. During execution, the memory consumed by live objects is constant, objects are generated at constant rate, and the shape of the heap (either a deep list of payload objects, or a shallow array of object references) does not change.

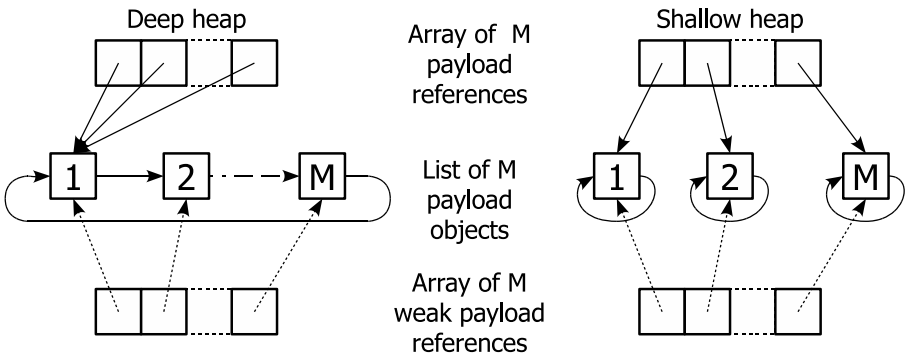


Figure 3.2: Heap Depth workload

The object structure used by the workload is depicted in Figure 3.2. The workload allocates a predefined number of payload objects and an array for holding root references to the objects. The payload objects are arranged in two different ways depending on the desired heap depth. In deep heap configuration, the allocated objects form a singly linked list, and all the elements of the array of root references point to the first object in the list. This results in average distance of every object from the root being equal to half the number of objects. In shallow heap configuration, each payload object refers only to itself and all references to the payload objects are kept in the array of root references. This results in the average distance of every object from the root being equal to one. To facilitate random object selection with constant complexity, weak references to all payload objects are kept in a separate array.

The workload repeatedly selects a random payload object and replaces it with a newly allocated object, without changing the object graph topology. As in the previous workload, the rate of object replacement is regulated by performing certain amount of phony work, in this case implemented as accessing a predefined number of randomly selected payload objects.

The number of payload objects, the depth of the heap, and the amount of phony work to be done can be set for each instance of the workload, and remain constant during workload execution.

Some experiments use an alternative version of the Heap Depth workload, which differs slightly from the version presented above. A doubly linked list was used in place of a singly linked one, and the list was present in both the shallow and the deep configuration of the workload. The change in average distance from root was achieved only by modifying the array of root elements. The reasons for this difference are purely historic.

3.1.1.3 Heap Size

The heap size workload was designed for experiments quantifying dependencies in GC behavior with respect to number of heap objects. During execution, the memory consumed by live objects is constant, garbage is generated at a constant rate, and the shape of the heap changes randomly.

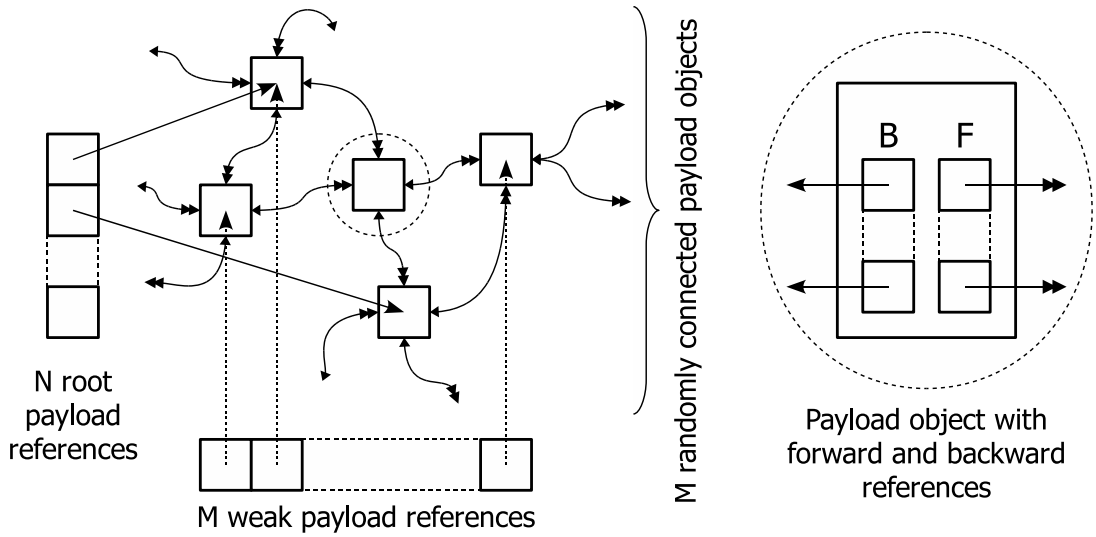


Figure 3.3: Heap Size workload

The object structure used by the workload is depicted in Figure 3.3. The workload allocates a predefined number of payload objects on the heap and arranges them in a random directed graph by connecting each object to a predefined number of randomly selected successors. Each payload object contains a list of *forward references* to its successors as well as a list of *backward references* to its predecessors. To provide the collector with entry references into the graph, a small number of *root references* to randomly chosen payload objects is kept in a separate array. To facilitate random object selection with constant complexity, weak references to all payload objects are kept in a separate array.

The workload repeatedly removes a randomly selected object from the graph, allocates a new payload object, and inserts it into the graph by connecting it to a

predefined number of randomly selected successors. As in the previous workloads, the speed of object allocation and destruction is regulated by performing certain amount of phony work, implemented as taking a predefined number of steps in walking the payload object graph.

The number of payload objects, the number of successors, and the amount of phony work to be done can be set for each instance of the workload, and remain constant during workload execution. With random workload construction we cannot guarantee if some large part of the structure was not disconnected from the rest and became garbage. However, we can detect such situation and we eliminated runs when that happened from the experiment results.

3.1.2 Experiment Results

The experiments were conducted on a Dell PowerEdge 1955 machine with Dual Quad-Core Intel Xeon CPU E5345 2.33 GHz (Family 6 Model 15 Stepping 11), 32 KB L1 caches, 4 MB L2 caches, 8 GB Hynix FBD DDR2-667 RAM, running Gentoo Linux kernel 2.6.27 x86_64, Sun Java SE Runtime Environment build 1.6.0-11-b03, Java HotSpot VM build 11.0-b16. We use the default garbage collector, the parallel throughput collector described in Section 2.4.

Where unspecified, the workload parameters were set to default values. The default maximum heap size limit was set to 64 MB per workload, which helps achieve a reasonable heap occupation without very large workloads. The default payload size and the default allocation speed were set to resemble the same parameters observed in the *compile*, *db*, *derby* and *sunflow* workloads of the SPECjvm 2008 suite [68].

All the experiments were executed multiple times. Since it turned out that the variation between executions was minimal except for a few cases deserving special attention, we have decided to consistently plot results from individual executions throughout the chapter. In results with minimal variation, the individual executions are representative, and in results deserving special attention, the difference between individual executions is illustrated and explained.

The collector overhead is measured using the diagnostic output of the GC implementation. The combined overhead is reported alongside separate values for the young generation and the tenured generation. The overhead is taken as an average from a stable 10 minutes long execution period and is calculated as a fraction of collection time over the total execution time (including collections).

3.1.2.1 Dependency on Number of Live Objects

The experiment to assess the dependency of GC overhead on the number of live objects uses the Heap Size workload. The number of live objects varies from 1 K to 64 K, which just about fills the available heap, all the other parameters are constant. The results in Figure 3.4 confirm a known fact that the GC overhead is especially significant under low memory conditions, but also suggest that the GC overhead stays relatively large even with more memory available. Furthermore, the results indicate a varying split of the overhead between the young generation collector and the tenured generation collector, an effect that is investigated later.

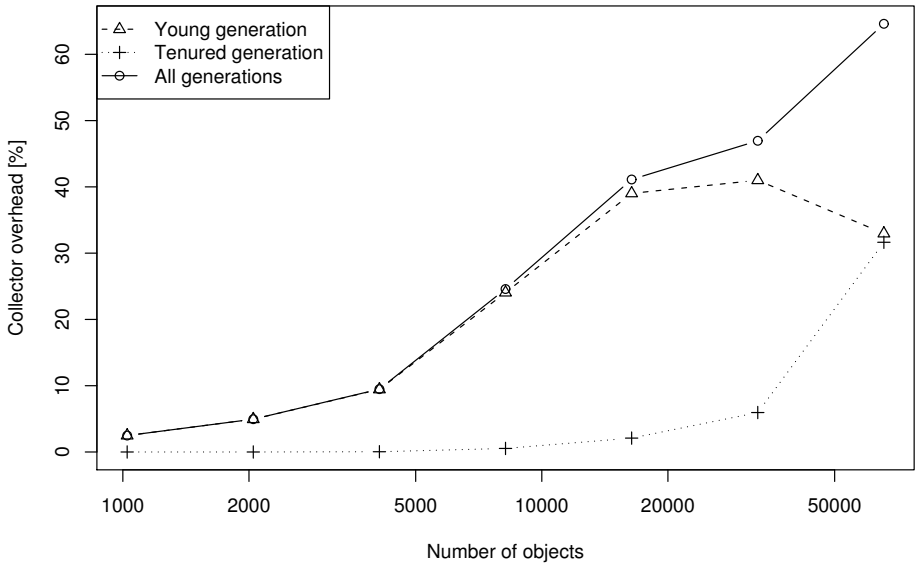


Figure 3.4: Dependency on number of objects

3.1.2.2 Dependency on Size of Live Objects

The experiment to assess the dependency of GC overhead on the size of live objects uses the Heap Depth workload. The size of live objects varies from 20 B to 100 B plus small constant overhead, the deep heap workload variant is used, all the other parameters are constant. The results in Figure 3.5, and similar results for other workloads not displayed here, suggest that there is very little dependency of GC overhead on the size of live objects for common allocation speeds. This effect shows how larger sizes lead to more frequent collections and more data copying on one hand, and to more tenured objects with less copying on the other hand.

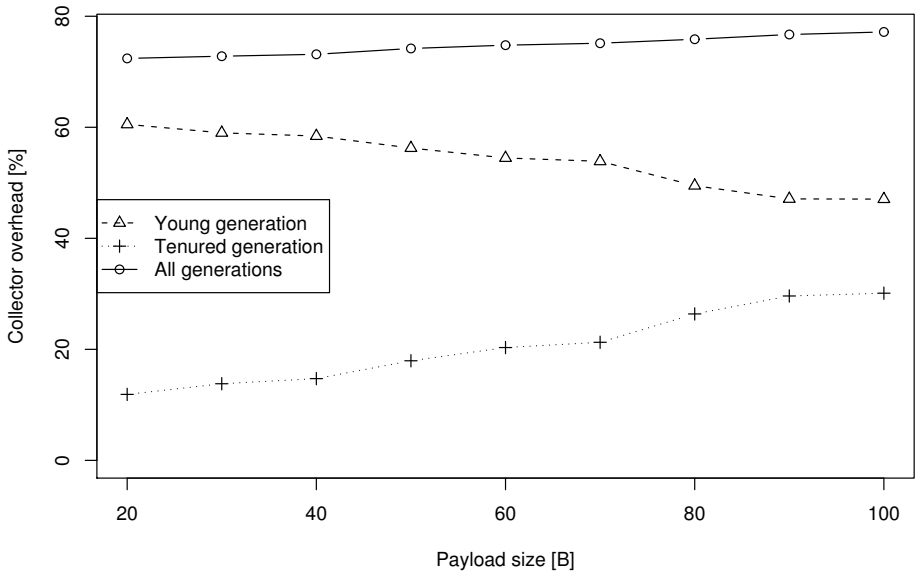


Figure 3.5: Dependency on size of objects

3.1.2.3 Dependency on Object Lifetimes

The experiment to assess the dependency of GC overhead on the age of live heap objects uses the Object Lifetime workload. The total number of components varies from 256 to 16 K, which just about fills the available heap, all the other parameters are constant. The results in Figure 3.6 are for the configuration with mostly old objects, the results in Figure 3.7 are for the configuration with mostly young objects. To explain the results, it is necessary to supplement the figures with information on generation sizes. For 16 K components and the configuration with mostly old objects, the GC generation sizing heuristic sets the eden space at 21 MB, the survivor spaces at 128 KB, and the tenured space at 43 MB. With mostly young objects, the eden space is at 7 MB, the survivor spaces are at 7 MB, and the tenured space is at 43 MB. The heuristic appears to be confused when the workload does not meet the generational hypothesis and only young objects are present, shrinking the eden space too much and leading to large overhead.

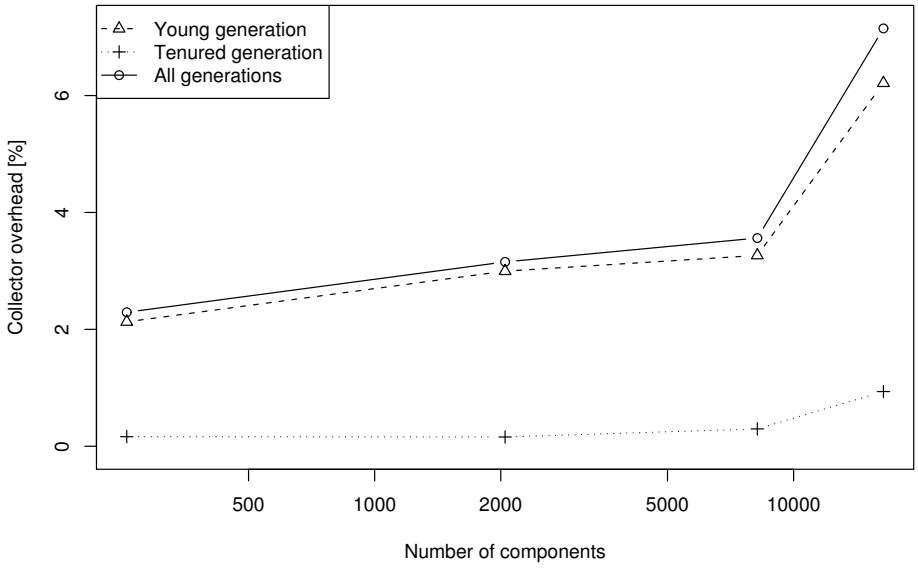


Figure 3.6: Object Lifetime, live objects mostly old

3.1.2.4 Dependency on Heap Depth

The experiment to assess the dependency of GC overhead on the depth of the heap object graph uses the Heap Depth workload. The number of live objects varies from 2 to 128 K for both the deep and shallow heap configurations, all the other parameters are constant. The results in Figure 3.8 are for the configuration with deep heap object structure, the results in Figure 3.9 are for the configuration with shallow heap. The results show higher GC overhead for the shallow heap configuration, which is unexpected.

We suspect the cause to be connected to the processing of remembered sets, which hold the references pointing from older to younger generations. Since the array of root references should be stored in tenured generation space and all the references it contains point to the young generation space containing the payload objects, the references cross a generation boundary and therefore belong

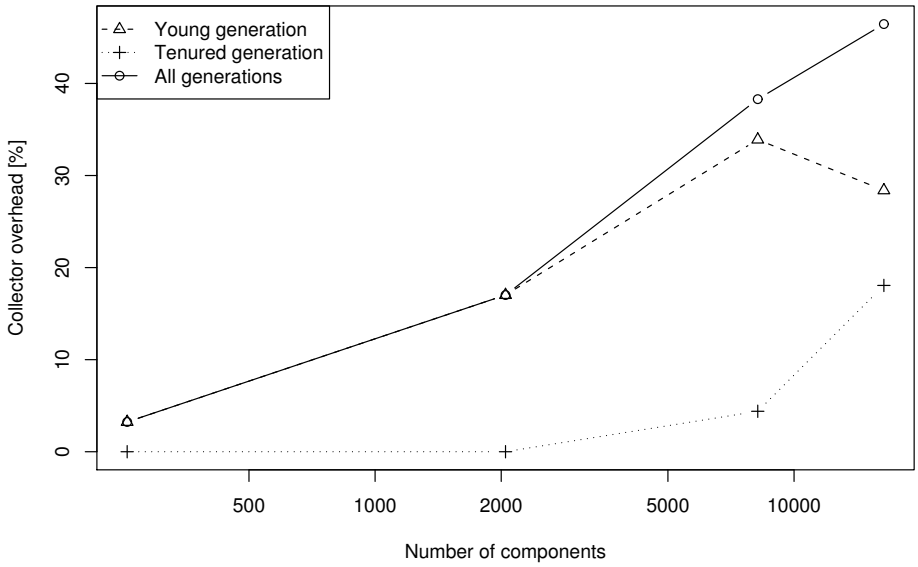


Figure 3.7: Object Lifetime, live objects mostly young

to the remembered sets. This in turn increases the number of roots that must be processed during young generation collection.

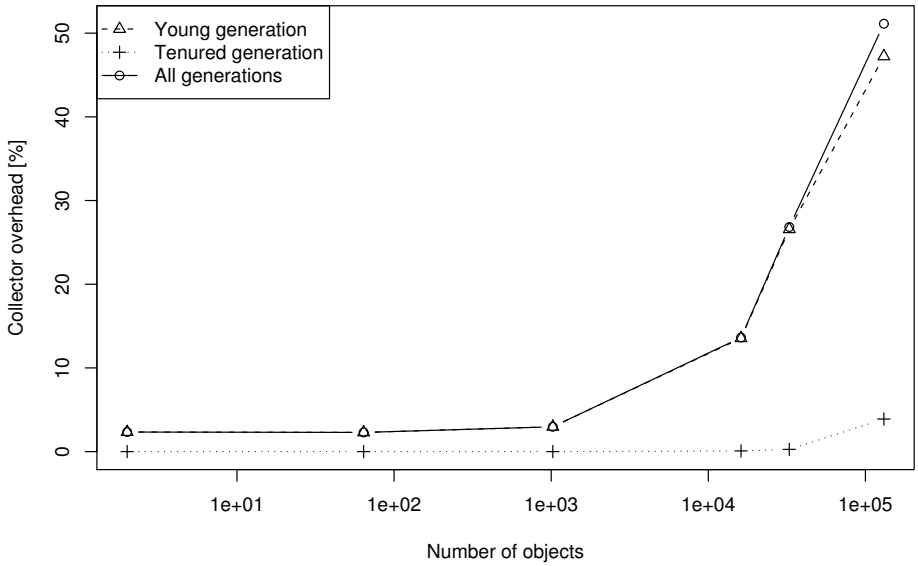


Figure 3.8: Heap Depth, deep heap

Our earlier experiments with the alternative Heap Depth workload have also revealed an interesting instability in the GC overhead between consecutive JVM executions. The instability is illustrated in Figure 3.10. In contrast to results shown in Figure 3.8, the GC overhead rose sharply at around 10 live objects, and continued to slowly increase with the number of live objects, save for the occasional drop after the first rise. The cause of the unexpectedly high GC overhead lies with the GC heuristic, which promoted some of the short-lived payload objects to tenured generation, resulting in significant increase of the GC overhead. The heuristic does not work consistently for small numbers of objects, hence the instability between consecutive JVM executions.

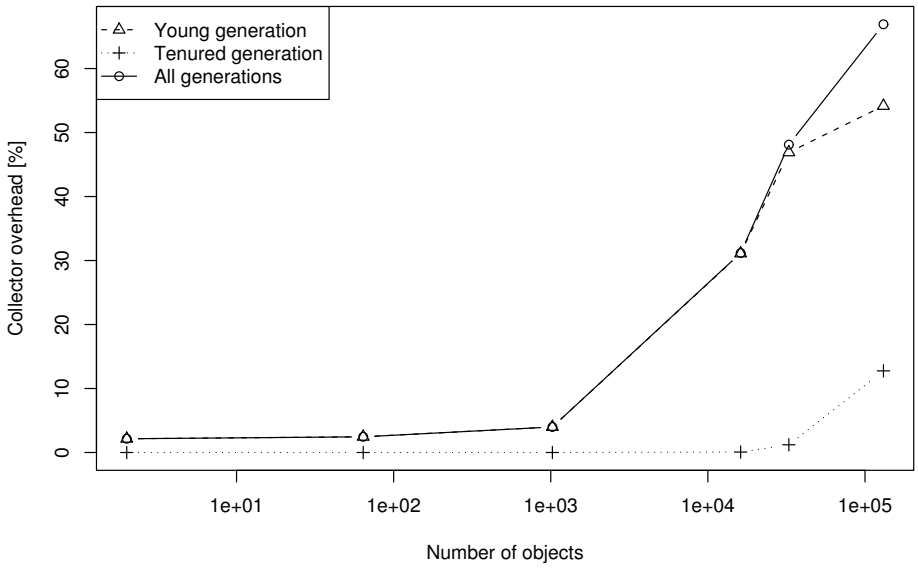


Figure 3.9: Heap Depth, shallow heap

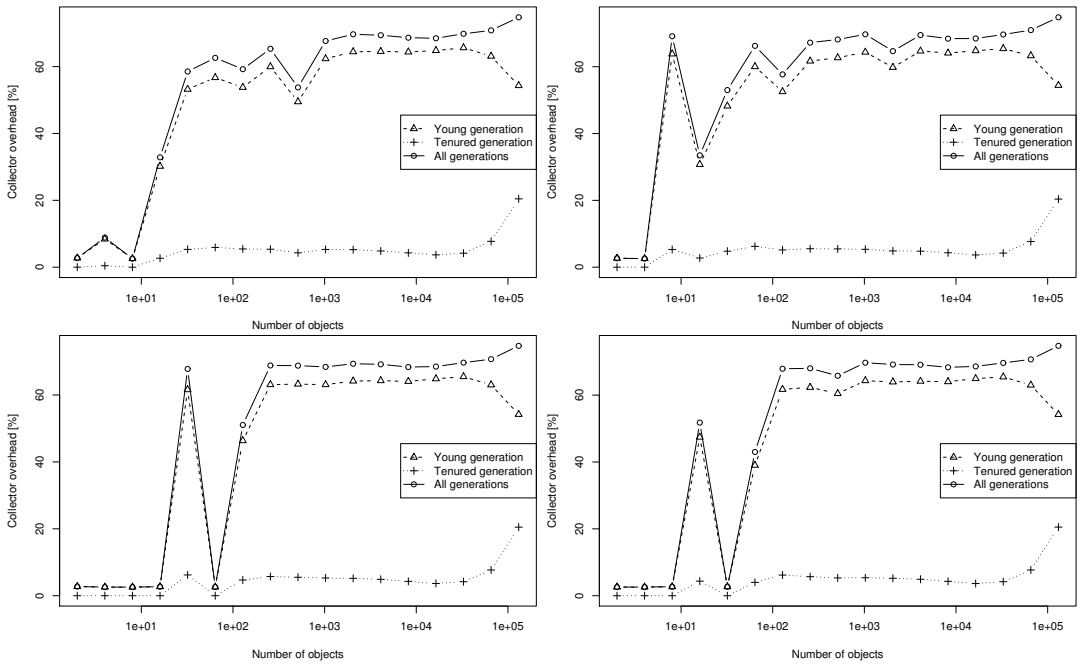


Figure 3.10: Four runs of Heap Depth, deep heap

3.1.2.5 Dependency on Object Allocation Speed

In the previous experiments, the speed of object allocation (and thus the speed of garbage generation) was kept constant while varying the number of live objects. The purpose of the next experiment is to assess the dependency of GC overhead on the object allocation speed. The experiment uses the Heap Size workload and the alternative Heap Depth workload with deep heap configuration. As mentioned earlier, the speed of object allocation can be adjusted by the amount of phony work performed between operations allocating new and releasing old objects. Therefore during execution, the experiment progressively changes the amount of phony work, resulting in varying allocation speed which is measured. The other parameters are kept constant.

The results in Figure 3.11 are for the Heap Size workload, the results in Figure 3.12 are for the Heap Depth workload with deep heap. Since the allocation speed translates directly into the collection rate, the overhead is expected to increase. We observe that for the young generation space, the GC overhead per object varies with allocation speed, while for the tenured generation space, the GC overhead remains constant. This effect can be attributed to the GC generation sizing heuristic, which in this experiment influences the young generation collector but not the tenured generation collector.

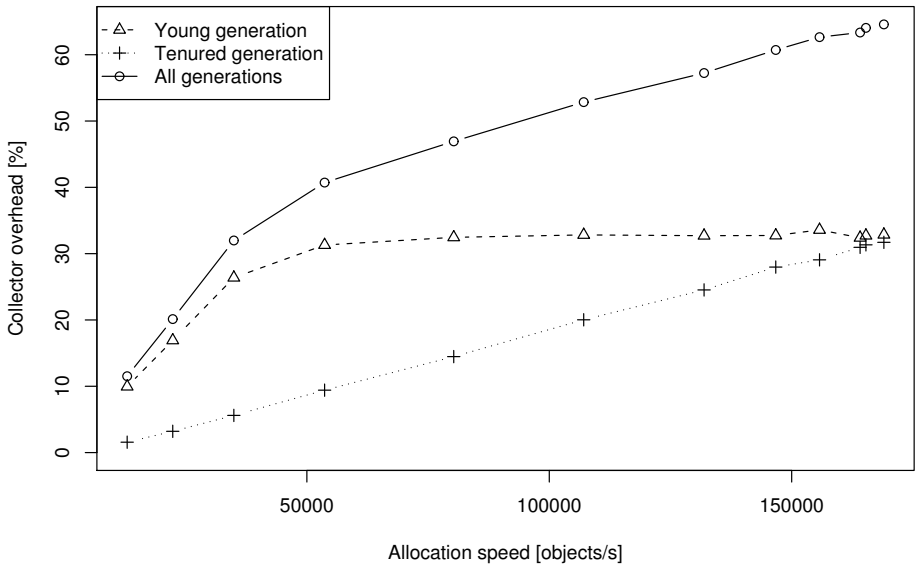


Figure 3.11: Allocation speed dependency, Heap Size

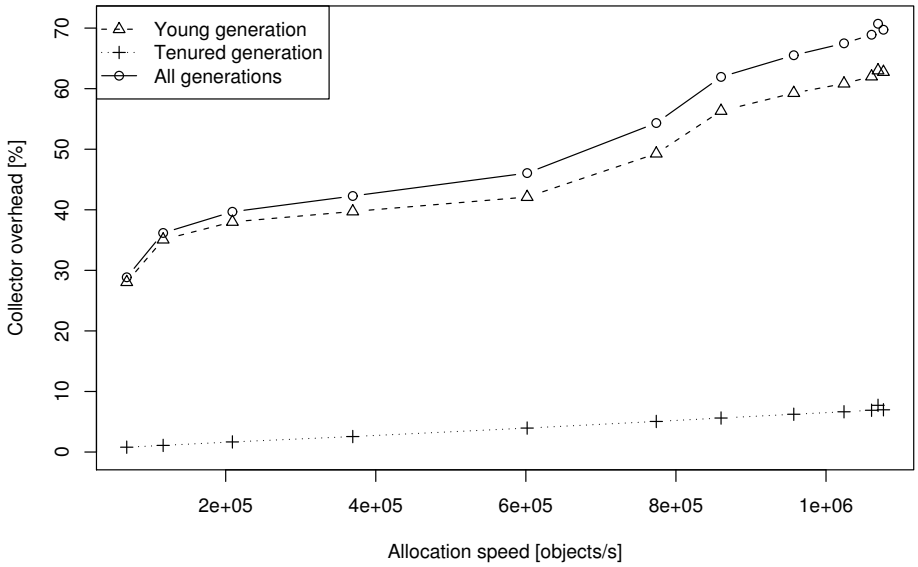


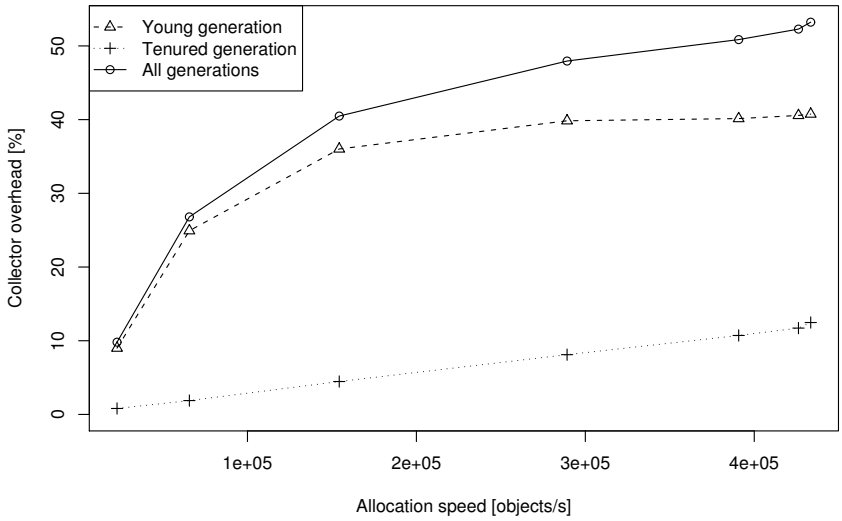
Figure 3.12: Allocation speed dependency, Heap Depth, deep heap

3.1.2.6 Dependency on Garbage Structure

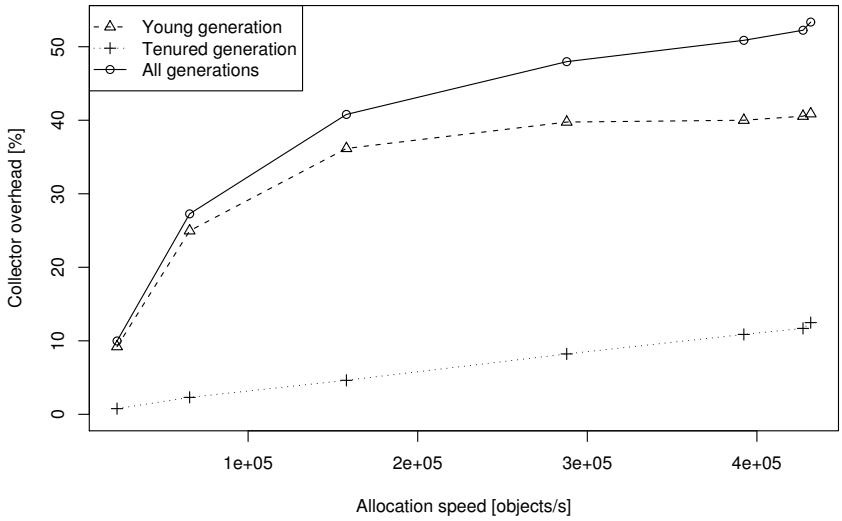
The experiment to assess the dependency of GC overhead on the structure of the garbage combines the Heap Size workload with the alternative Heap Depth workload. The combination of the workloads is implemented by letting each of the workloads allocate a predefined fraction of the (constant) total number of live objects before switching to the other workload. The ratio of the two fractions determines the ratio of the workload combination. This gives rise to multiple experiment instances with different workload ratios. During execution of one experiment instance, the allocation speed varies, while the other parameters are kept constant.

The results in Figure 3.13 plot the dependency of GC overhead on allocation speed under different workload ratios. The dependency differs from that of the individual workloads found in Figures 3.11 and 3.12, and is the same for all tested workload ratios, of which we only show a few for brevity. From this, we can conclude that the structure of garbage on the heap does not influence the GC overhead, which is to be expected, because the collection algorithms do not visit unreachable objects at all.

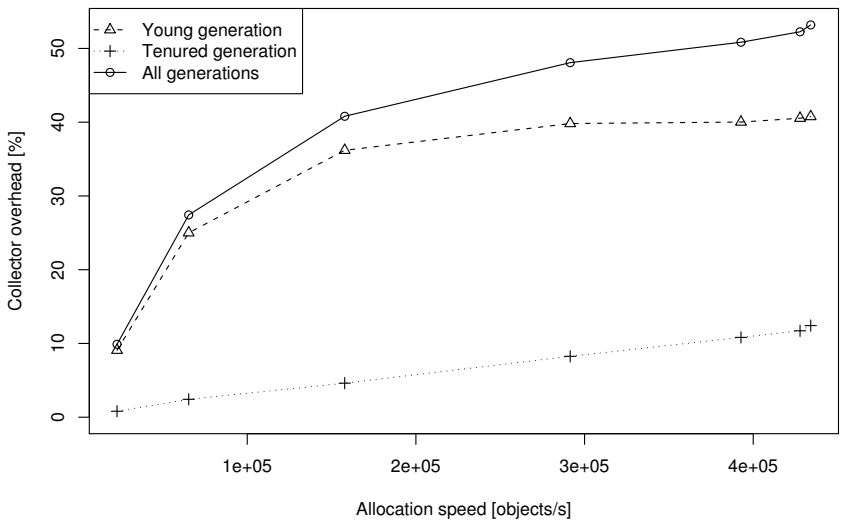
Each experiment instance was executed in 4 different runs (JVM restarted), with the coefficient of variation between values from different runs of the same experiment instance staying under 1 % in 90 % of measurements, with the maximum of 7.4 %.



(a) Ratio 3:1 in favour of Heap Size workload



(b) Ratio 1:1



(c) Ratio 3:1 in favour of Heap Depth workload

Figure 3.13: Dependency on allocation speed with different ratio of allocations from Heap Size and Heap Depth workload.

3.1.2.7 Dependency on Maximum Heap Size with Constant Heap

This experiment goal is to assess the dependency of GC overhead on the maximum heap size available to the virtual machine (adjustable using the -Xmx command line parameter). Previous work [15] states that collector overhead should decrease with increased heap size limit, roughly like the $1/n$ function. We run the experiment in four configurations: the Objects Lifetime workload configured with live objects mostly old, the Heap Depth workload with deep heap, the Heap Size workload configured for fast allocation speed and the last one is again Heap Size workload configured for slow allocation speed.

During execution of one experiment instance, the maximum available heap size varies from 64 MB to 2048 MB, while the other parameters are kept constant.

The results are in Figures 3.14-3.17. The results for the Objects Lifetime and Heap Depth workloads are surprising—the collector overhead is growing after initial decrease, which is unexpected. This might be caused by incorrectly applied heuristics that are sizing the heap generations. With both variants of Heap Size workload we can observe significant decrease in GC overhead with increasing maximum memory available to the heap. This decrease is more pronounced in case of the fast-allocating workload configuration.

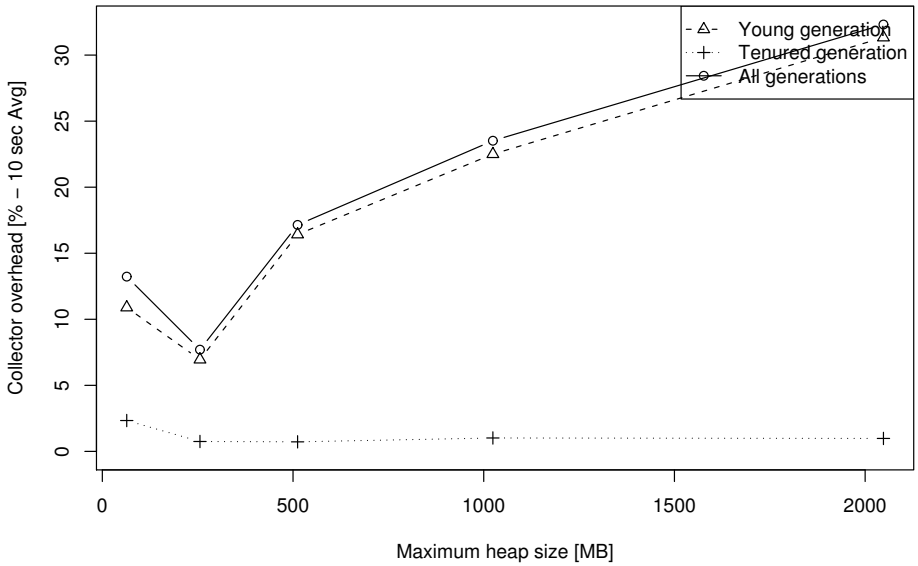


Figure 3.14: Dependency on maximum heap size, Objects Lifetime – mostly old objects

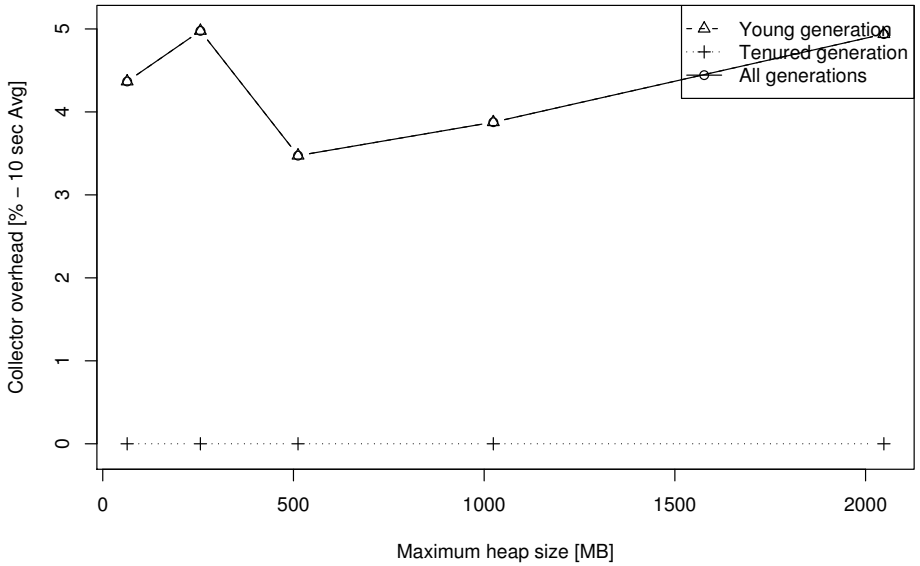


Figure 3.15: Dependency on maximum heap size, Heap Depth, deep heap

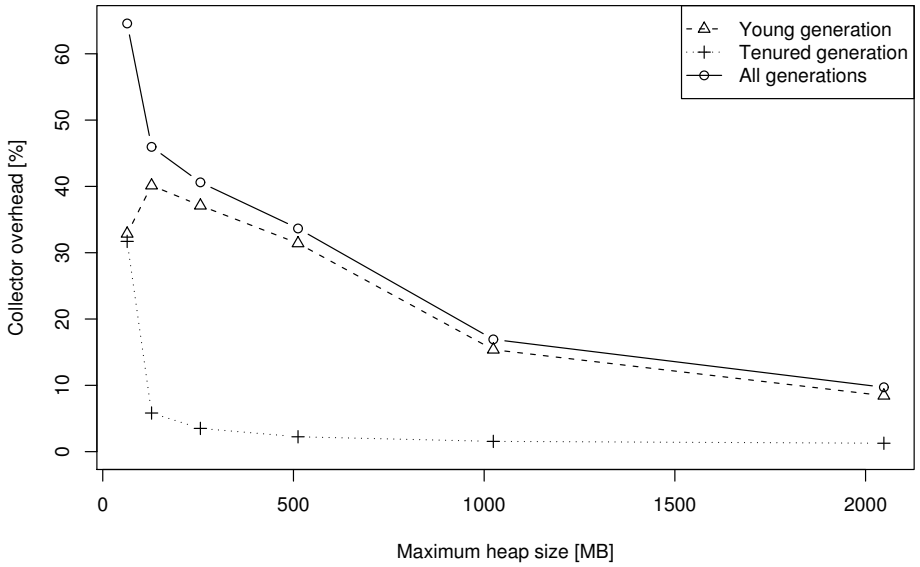


Figure 3.16: Dependency on maximum heap size, Heap Size, fast allocation

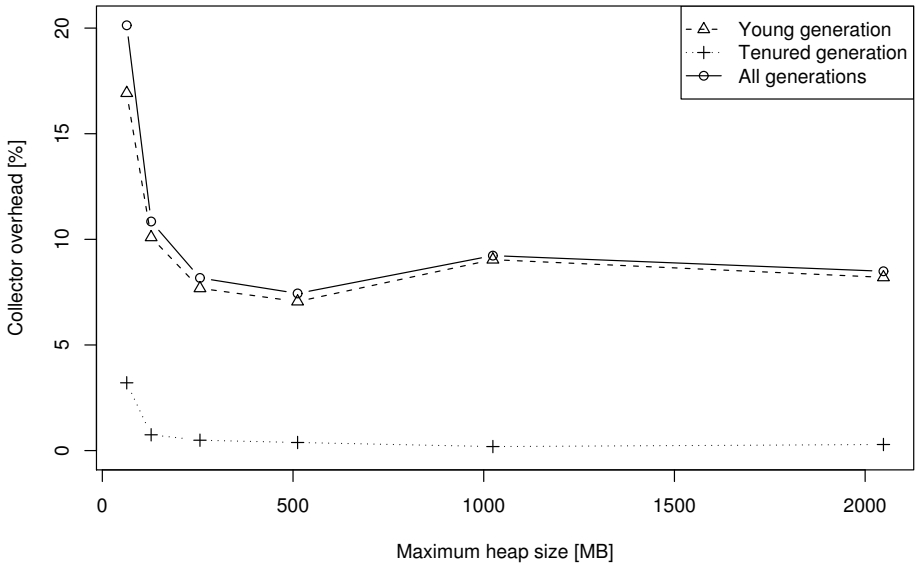


Figure 3.17: Dependency on maximum heap size, Heap Size, slow allocation

3.1.2.8 Dependency on Maximum Heap Size with Growing Heap

The last experiment is a variation of the previous experiment. The aim is to assess the dependency of GC overhead on the maximum heap size, but this experiment uses different workloads and varies an additional parameter during execution. The experiment uses either the Heap Size workload or the alternative Heap Depth workload with deep heap configuration. The maximum available heap size varies from 64 MB to 2048 MB, the heap size represented by live objects varies at the same rate as the maximum heap size, all the other parameters are kept constant. The ratio of maximum heap size to the size of live objects was approximately 4:1.

The results in Figure 3.18 are for the Heap Size workload, the results in Figure 3.19 are for the Heap Depth workload with deep heap. In contrast to the previous experiment with constant heap size, we can observe that the GC overhead remains approximately constant when we also increase the amount of memory consumed by live heap objects at the same rate as the maximum heap size. The time needed to perform a collection increases proportionally to the number of live objects, eliminating the decrease in overhead gained by increasing maximum heap size.

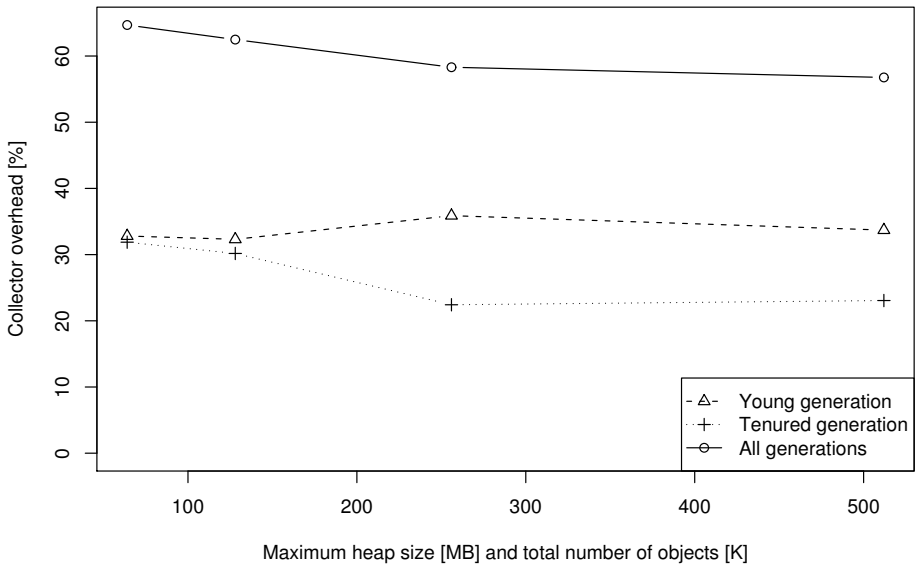


Figure 3.18: Dependency on maximum heap size and object count, Heap Size

In general, the decrease in GC overhead when increasing the maximum heap size should be an inherent property of the collector. If the time to perform a collection depends mainly on live objects, doubling the maximum heap size should also double the interval between two collections, because it takes twice the time to fill the heap (assuming constant allocation speed) and the time needed to perform each collection remains constant (assuming constant number of live objects).

As shown in previous work [15] and in our experiments, this assumption seems to be mostly valid. There are, however, certain workloads for which this assumption does not hold. One of the Object Lifetime experiments exhibited a steady overhead of about 40 % regardless of the increase in maximum heap size. Our working hypothesis is that in workloads with too many young objects, the generation sizing heuristic causes some of the young objects leak into the tenured generation, which brings a significant increase in overhead.

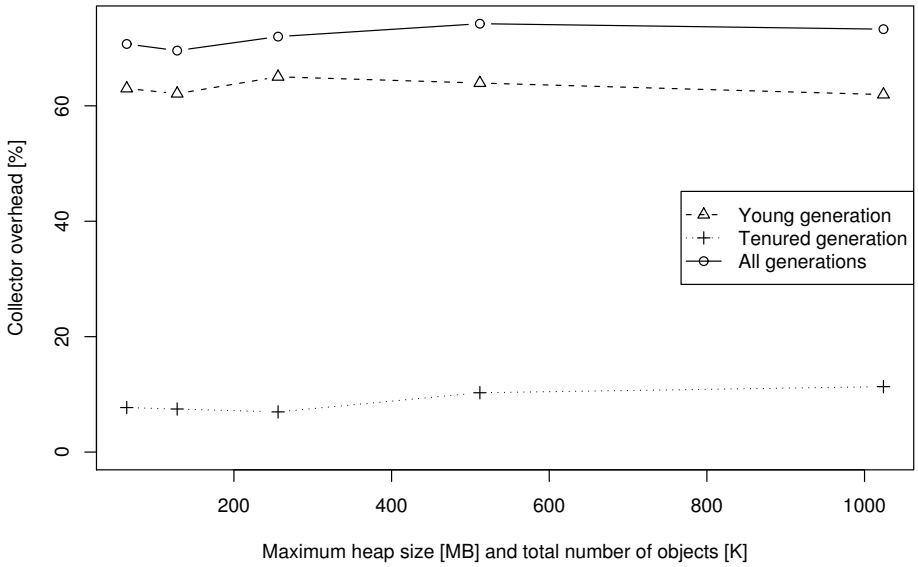


Figure 3.19: Dependency on maximum heap size and object count, Heap Depth, deep heap

3.1.3 Comparison with IBM Virtual Machine

From the experiment results above, it is obvious that the collector performance is hard to grasp using simple functions or dependencies. To compare the behavior of the relatively complex two-generation collector with a much simpler one-generation collector, we repeated some of the experiments on the J9 virtual machine from IBM, which has a simple stop-the-world mark-sweep collector with only one generation and occasional compaction.

We used the same machine as in the experiments with Oracle's HotSpot virtual machine, running IBM J9 VM (build 2.4, J2RE 1.6.0) with collector optimizing throughput (default, or -Xgcpolicy:optthruput option).

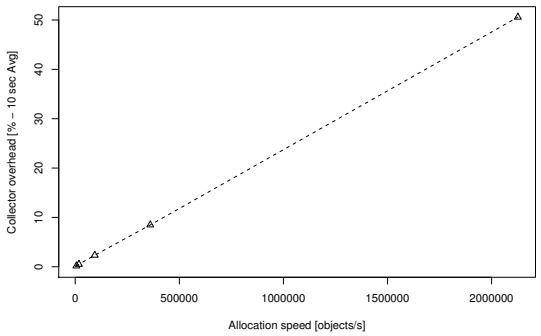
3.1.3.1 Dependency on allocation speed

We start with allocation speed experiments, counterparts to Section 3.1.2.5. The results from HotSpot are in Figures 3.11 and 3.12, where it is difficult to find some tendencies or patterns. In contrast, multiple results for the IBM VM, displayed in Figure 3.20, all have a clear linear tendency which is what one would expect—allocating twice as fast will cause the heap is filled twice as fast and given the size of live data is the same it is expected the individual collections will take the same time and therefore the overhead should be twice as big.

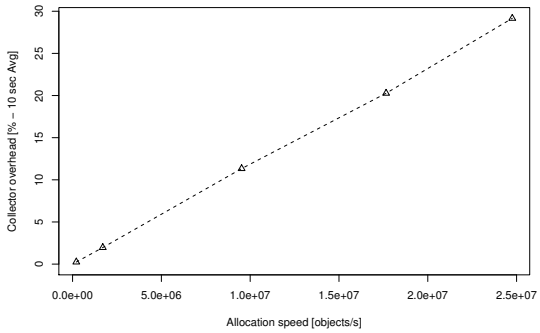
3.1.3.2 Workload combination—dependency on garbage

This is the IBM VM counterpart to Section 3.1.2.6. We show the results for the same workload combination, Heap Size and Heap Depth.

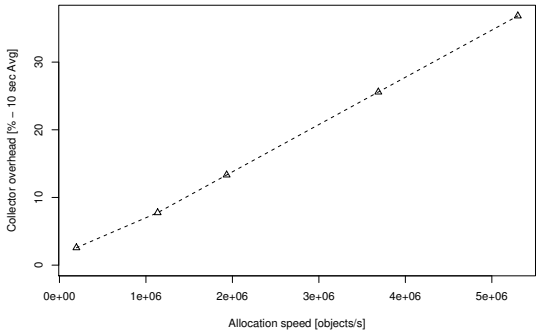
The results for the deep configuration of the Heap Depth workload are shown in Figure 3.21, the HotSpot equivalent is in Figure 3.13. On the HotSpot JVM, the results show there is no difference in overhead with different allocation ratios and we can conclude the collector overhead does not depend on the garbage on the heap. In contrast, the results from the IBM JVM show a small decrease of



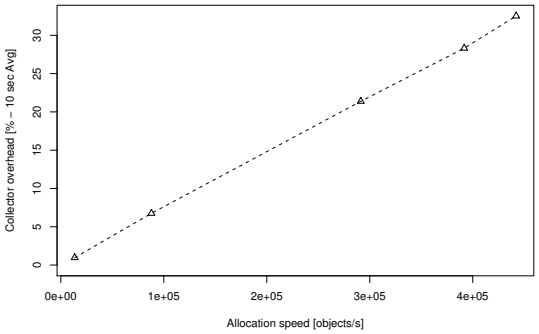
(a) Heap depth workload – deep variant with 128K objects



(b) Objects lifetime workload - old objects variant with 1 component



(c) Objects lifetime workload - young objects variant with 16K component



(d) Heap size workload with 64K objects

Figure 3.20: Dependency of the collector overhead on allocation speed, IBM VM

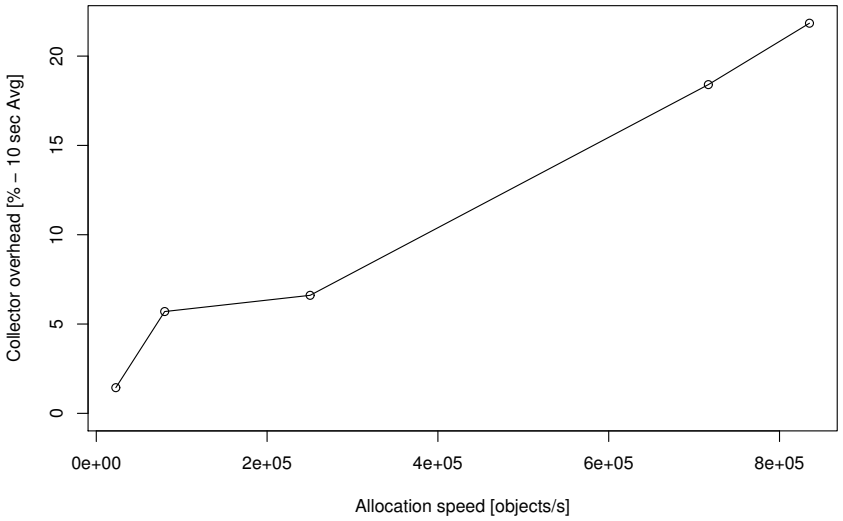
overhead when the experiment allocates more objects in the Heap Depth workload. The Heap Depth workload in isolation has lower overhead from the two workloads in the combination (Figures 3.20a and 3.20d), suggesting it might be caused by the workload mix. However, the decrease is not big enough to state that there is a visible dependency on the garbage for that collector.

3.1.3.3 Dependency on maximum heap size

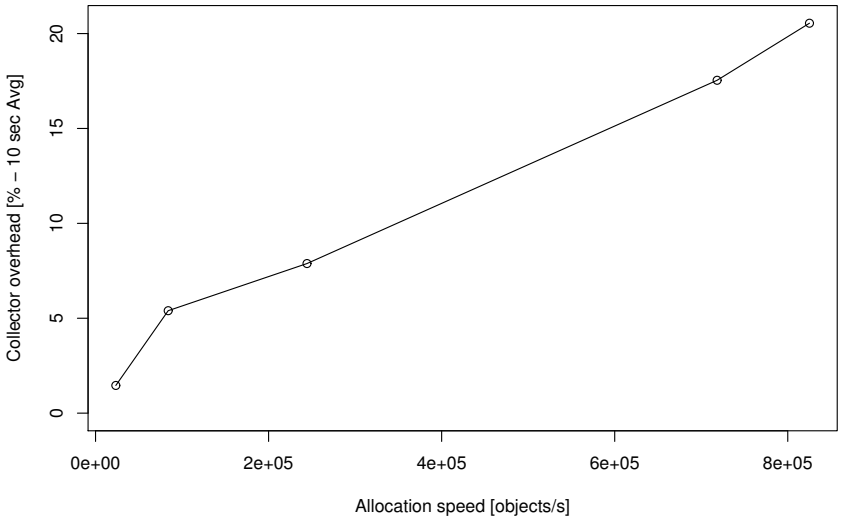
This is to complement Section 3.1.2.7 with the results from the IBM VM.

The results for three selected experiment instances are shown in Figure 3.22. Unlike HotSpot VM, the collector overhead decreases in a manner close to the expected $1/n$ function.

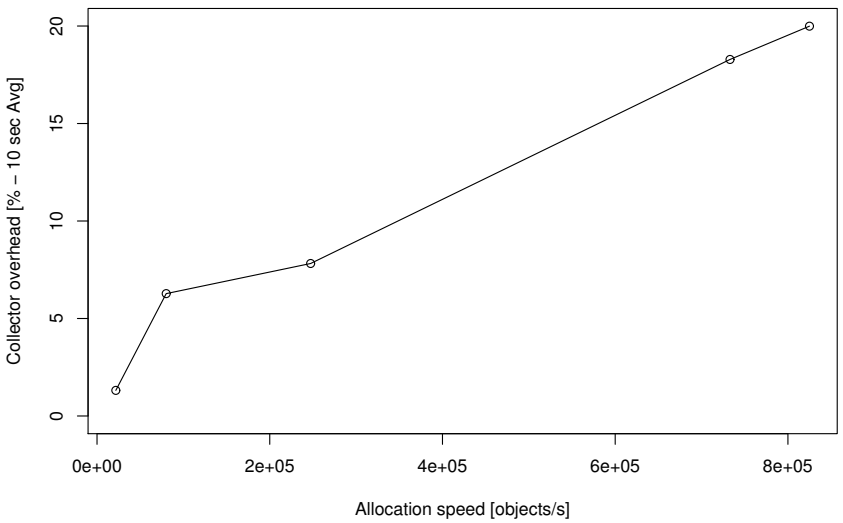
Overall, the collector in the IBM VM is much more simple than the one we are using in the HotSpot VM, therefore it is no surprise the behavior is much more predictable. This shows us the difference between performance of different collector is not different only in terms of effectivity but also it has different performance patterns and leads us to finding it will it is unlikely the black-box approaches will be successful.



(a) Ratio 3:1 in favor of Heap Size workload

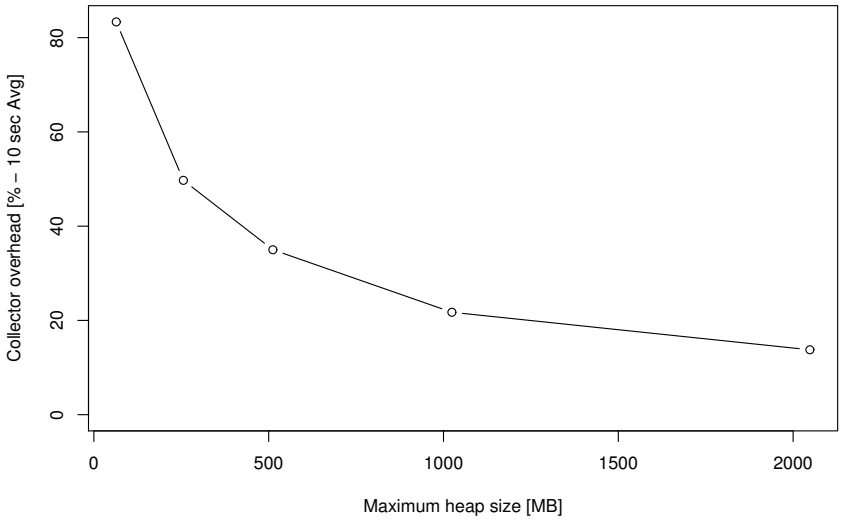


(b) Ratio 1:1

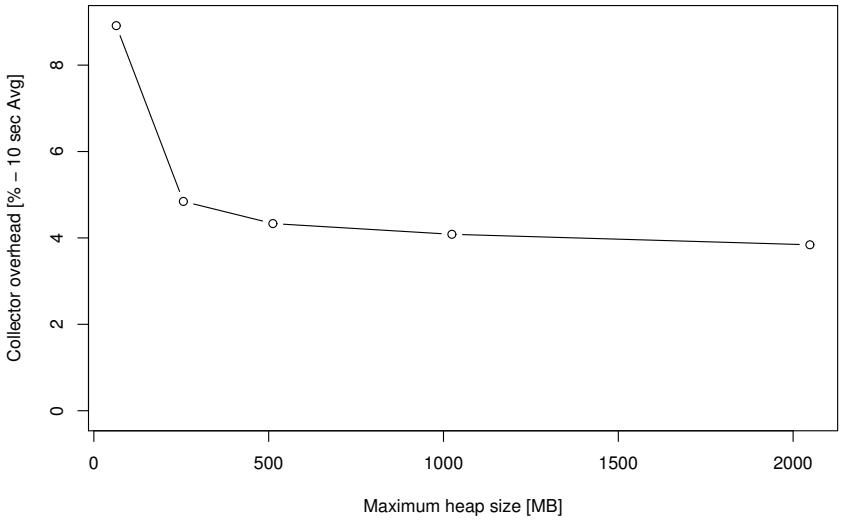


(c) Ratio 3:1 in favor of Heap Depth workload

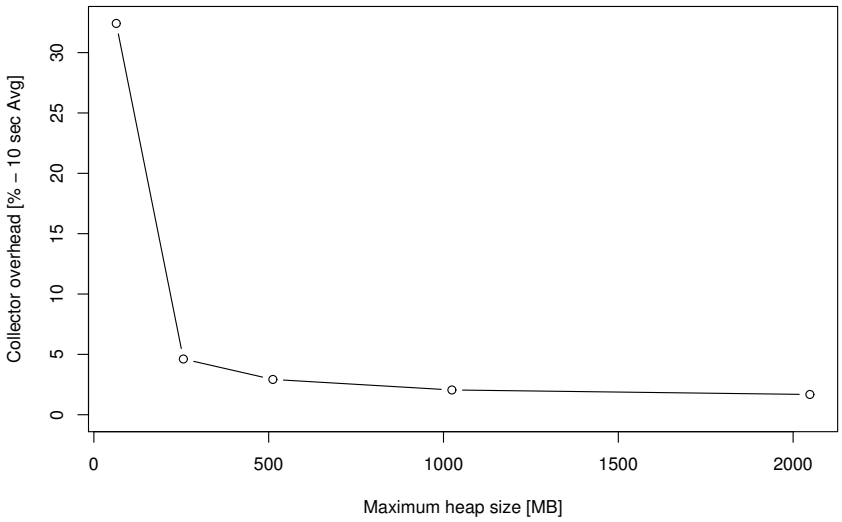
Figure 3.21: Dependency of the collector overhead on allocation speed in combined workload and different ratios of allocations in single workloads, IBM VM



(a) Objects Lifetime workload, old objects



(b) Heap Depth workload, deep heap



(c) Heap size workload, fast allocation

Figure 3.22: Dependency of the collector overhead on maximum heap size, IBM VM

3.2 Observed Issues

As pointed out in the beginning of the chapter, the experiment results presented here apply only to the specific GC implementation and therefore do not allow us to make general observations. Due to considerable differences in results from two GC implementations, observations valid for multiple GC implementations are only very general. We can, however, provide answers to the questions raised in the beginning of this chapter, especially where performance modeling, component benchmarking, and performance optimization are concerned.

3.2.1 Modeling

The fact that the GC overhead can easily reach one third of the application execution time even without excessive allocation speeds and excessive memory shortage provides a clear motivation for further work on including garbage collection in performance modeling. Coupled with this, however, is the sobering observation that even simple code patterns, such as creating deep lists on the heap, can trigger fluctuations in the GC overhead that reach as much as half of the application execution time. It is unlikely that a performance model would predict such fluctuations, suggesting the existence of modeling precision limits due to the instability of the GC overhead.

Among the workload parameters that bear strong influence on the GC overhead are the allocation speed, directly related to the collection rate, and the object lifetimes, directly related not only to the efficiency of generational garbage collection under a varying degree of validity of the generational hypothesis, but also to the ability of the generation scaling heuristic to set the eden space and survivor space sizes correctly. These parameters need to be specified as attributes of the individual components making up the performance models.

3.2.2 Benchmarking

The presence of GC overhead also needs to be factored into the way benchmarks are constructed. This does not concern the application level benchmarks as much as the component level benchmarks—when a benchmark exercises an entire application correctly, the GC overhead is simply present alongside all the other performance related effects that the application would observe. When a benchmark exercises only a single component, the GC overhead present during the benchmark might not correspond to the GC overhead observed when the component is integrated into an application. The experiment results indicate that small heap sizes and high allocation speeds (both of which are likely to occur in a benchmark that exercises a single component repeatedly) typically lead to observing unrealistic GC overhead.

The component level benchmarks are frequently executed to populate performance models with attributes of the individual components. When this is the case, the timing information produced by the benchmarks is further combined to calculate the timing characteristics of the entire application. Since the execution time spent in calculation adds up differently than the execution time spent in collection, these two attributes should be reported separately, accompanied by

basic workload parameters that bear influence on the GC overhead, as outlined above.

3.2.3 Optimization

The experiment results presented in this chapter underscore the fact that the GC overhead is sensitive to low memory conditions—not only when the entire heap is almost used up, but also when the size of a single generation approaches the space allocated to it by the generation scaling heuristic. Per generation memory usage statistics should therefore be added to the factors used in determining various platform parameters related to load balancing, such as the optimum thread pool sizes [52].

3.3 Summary of Chapter 3

We have carried out a series of experiments to familiarize ourselves with the issues to expect when modeling application performance in environments with garbage collection (we have shown a subset of results, more are available in [4]). What is different in our experiments compared to common garbage collection evaluation measurements (i.e. [15]) is that we do not subject the virtual machine to a load resembling real application workload, rather we stress it in a specific way in an attempt to expose performance patterns and parameters.

Unfortunately, the results give us little hope to construct a parametric black-box model (like Happe et al. [33] did for the messaging middleware), based on a set of benchmarks that could be executed on the target platform and we would derive the performance model automatically. Therefore we will concentrate on a single collection algorithm (generation throughput collector from the HotSpot VM, described in detail in Section 2.4, which is the default collector for widely used Java platform implementation).

The results also persuade us that including garbage collection into performance models is needed. In some workloads, we observed overhead of tens of percents of total execution time even in settings with enough headroom for allocations. The overhead spikes and rapid performance degradation are less likely in real application workloads, but significant differences between model results and measured performance can still occur.

Another lesson we learned from the results is that dynamic setting of the collector parameters, such as space sizes and tenuring threshold using ergonomics feature of the HotSpot VM can cause large performance fluctuations. To eliminate this, we will always fix generation sizes and tenuring threshold and also turn off the adaptive sizing policy to have more predictable results. This may be considered limiting, but current best practices on garbage collection tuning recommend fixing the sizes anyway [40, 42].

Chapter 4

Limits of GC Performance Modeling

After analyzing our findings from the previous chapter, we decided to fully concentrate our effort on a single collector—the parallel throughput collector from the HotSpot VM. We've also given up the attempts to construct a model with garbage collector as a black-box and now define the models based on the knowledge of the algorithms the collector is using, investigating what precision we can expect.

This chapter is based on following paper:
[49] P. Libič, L. Bulej, V. Horký, and P. Tůma. On the limits of modeling generational garbage collector performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 15–26. ACM, 2014

[49]

Assuming the perspective of an application developer with knowledge of GC principles—but very limited influence on particular GC internals—our goal in this chapter 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. The main points investigated in this chapter are 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 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.

This chapter is structured as follows: in Section 4.1, we present the general approach applied to modeling one-generation GC in Section 4.2 and two-generation GC in Section 4.3. We analyze model sensitivity to reduced and inaccurate input in Sections 4.4 and 4.5, respectively, and summarize the findings in Section 4.6.

4.1 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 [65], or total bytes allocated [36].

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 experience and experiments suggest that the duration of individual collections is often in an approximately linear relationship with the number of objects surviving the collection, but only within a single workload.

4.2 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,

(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 the size and lifetime (number of allocations until the 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:

$$HS = \left(\sum_{j=n_{i-1}+1}^{n_i} SIZE\,[j] \right) + \left(\sum_{\substack{j \in \{1 \ldots n_{i-1}\} \\ DEATH[j] \geq n_{i-1}}} SIZE\,[j] \right) \tag{4.1}$$

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 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 Equation 4.1 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 Equation 4.1. 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

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

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 Equation 4.1 into:

$$n_i - n_{i-1} = \frac{HS}{OS} - LT \tag{4.2}$$

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).

4.2.1 Model Evaluation

Although Equation 4.2 is fairly simple, the potential loss of accuracy introduced by averaging is difficult to estimate analytically. We have therefore validated Equation 4.2 experimentally for the DaCapo 2006.10 benchmark suite [17] running on the Jikes RVM 3.1.0 with the BaseBaseSemiSpace configuration [3].

We have chosen the Jikes RVM mainly because it makes it relatively simple to measure object allocations without the need for bytecode instrumentation (and the relatively large infrastructure it needs). The allocations in Jikes are performed directly by calling a single managed heap method. In production virtual machines, like Oracle's HotSpot, there is no single allocation entry point, because allocations are directly compiled into executed code with JIT. We modified this allocation method to measure the object sizes.

With Jikes, it is also possible and relatively straightforward to estimate average lifetime by sampling. We counted the numbers of live objects during garbage collections. Again, in the SemiSpace configuration, the collector is using a single method to perform object copying, so we instrumented this method to count live objects. Then we averaged the values from multiple samples we obtained.

For each benchmark, the results in Table 4.1 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 Equation 4.2 predicts fewer allocations between collections and vice versa.

Given the extreme simplicity of Equation 4.2, we consider these results promising—while not usable for accurate performance prediction, they suffice for better-vs-worse analysis and similar uses.

4.2.1.1 One-Generation Simulation

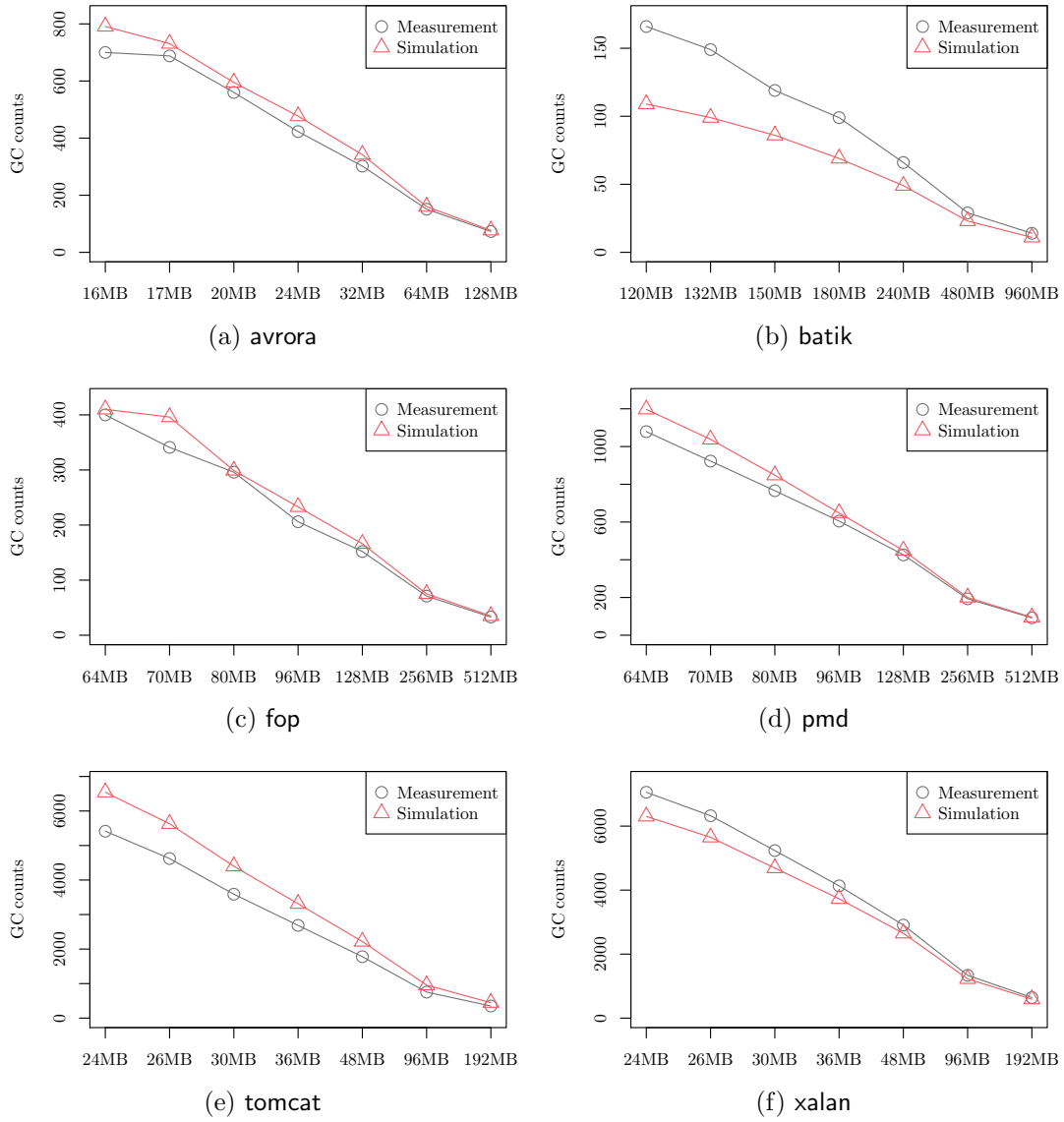
As a preparation for the two-generation collector, where we have to use a much more complex model, we also evaluated the model using simulation based directly on the formula in Equation 4.1. For that, we need a complete log of $SIZE[j]$ and $DEATH[j]$ or $LIFETIME[j]$ values. In other words, a trace of all object allocations along with their lifetimes.

A detailed description of how we implemented the tracing tool is given later, in Section 4.3.2, which is more complicated than the one we used for single-generation collector. In short, we have created a JVMTI agent and bytecode instrumentation, where the instrumentation intercepts all object allocation instructions and passes the allocated object references to the agent. It logs the object

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

Table 4.1: Collection intervals measured / predicted by Equation 4.2.

allocation and forces garbage collection in regular and short intervals (granularity) to observe what objects became unreachable. We use the JVMTI ObjectFree event for this—given the allocation and deallocation event in the log, we can calculate the approximate object lifetime. This is called *brute-force* approach in literature.



(a) avrora (b) batik

(c) fop (d) pmd

(e) tomcat (f) xalan

Figure 4.1: Measured and simulated garbage collection counts, IBM VM.

We have evaluated the simulation model using the IBM J9 virtual machine, build pxa6460sr8fp1-20100624_01(SR8 FP1). The reason for using this machine is that it has a stop-the-world one-generation collector and a working JVMTI implementation, which we need to trace objects that were collected. Unfortunately, getting this kind of information in Jikes RVM requires significant VM modifications meaning basically implementing the whole garbage collector from scratch. Oracle's HotSpot has no one-generation stop-the-world collector.

For evaluation, we used the DaCapo benchmarking suite, version 9.12-bach, with -n 100 and --no-pre-iteration-gc options. For lifetime tracing, we used following granularities: 1000 allocations for avrora and batik, 4000 for pmd, 5000

for xalan and tomcat and 10000 allocations for fop. The virtual machine is using a single-generation heap with the throughput collector (-Xgcpolicy:optthruput command line option), which uses the mark-sweep collection algorithm with occasional compaction. In order to get a more predictable behavior, we force the compaction after every collection. We measure the number of garbage collections and compare them with the number of collections predicted by the simulation model. The results are in Figure 4.1

The results are promising, again. We attribute the imprecision that remains to deficiencies of the model input collection, model simplifications or nondeterminism of the execution.

We now proceed with a similar investigation for a common two-generation collector.

4.3 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,

(c) GC stops the mutator,

(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 or mark-and-compact GC, its memory consists of one old space,

(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 [60].

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. If we also assume no fragmentation (k), the behavior of mark-and-compact and mark-and-sweep algorithms is identical. 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 both Serial and Parallel Throughput 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 4.1 and 4.2—these would turn out to be either overly complex or overly simplified. We therefore proceed by evaluating the model using a simulator.

4.3.1 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.

4.3.2 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 [36]—is both faster and more accurate, but also more complex and difficult to implement on widely used JVMs.

Because Elephant Tracks [65] (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 [54] and a custom JVMTI [61] 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.

4.3.3 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 [17]—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

[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.

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 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.

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

Due to large amount of experiments and results, in this and following two sections, we display in one plot data from multiple experiments. The legend to the plot labels is in Table 4.2, 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 without displaying excessive number of plots.

The plots in Figures 4.2, 4.3, 4.4, 4.5 and 4.6 show the young generation GC counts for the batik, fop, tomcat, xalan, 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*. In general the results show good accuracy with the exception of the multifop workload (Figure 4.6).

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.

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

Table 4.2: Plot legend labels

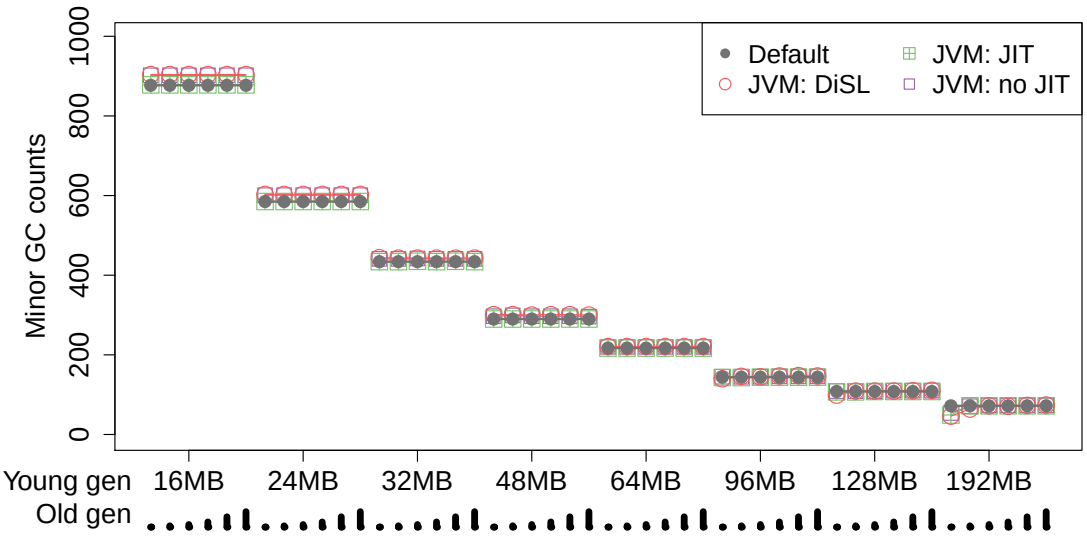


Figure 4.2: Young GC counts – JVM: batik

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 [65].

The plots in Figures 4.7, 4.8, 4.9, 4.10 and 4.11 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

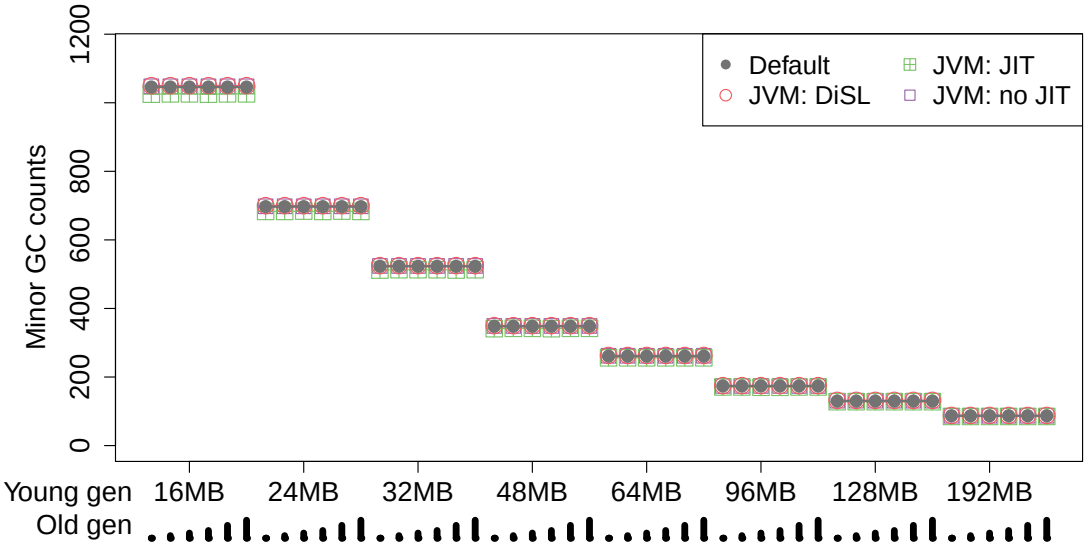


Figure 4.3: Young GC counts – JVM: fop

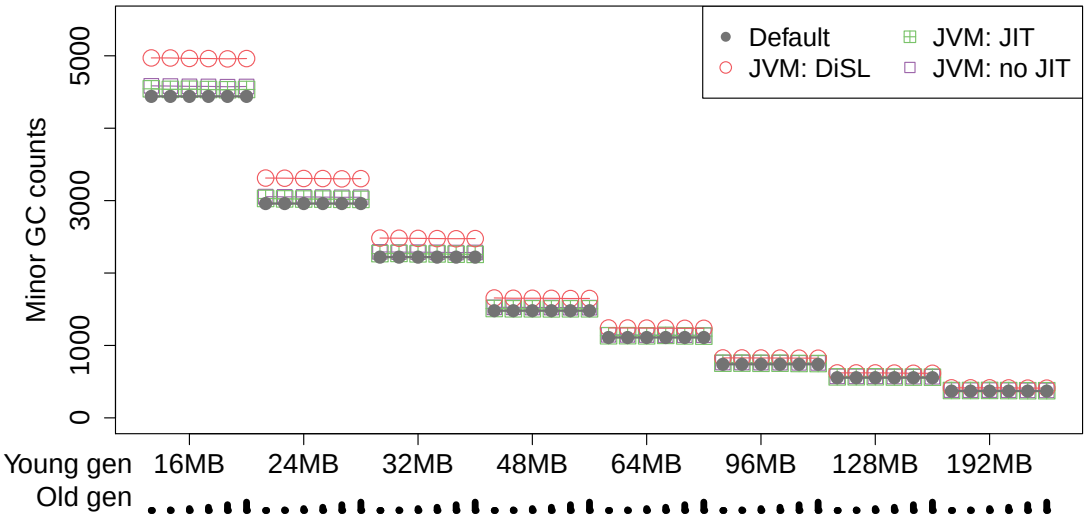


Figure 4.4: Young GC counts – JVM: tomcat

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.

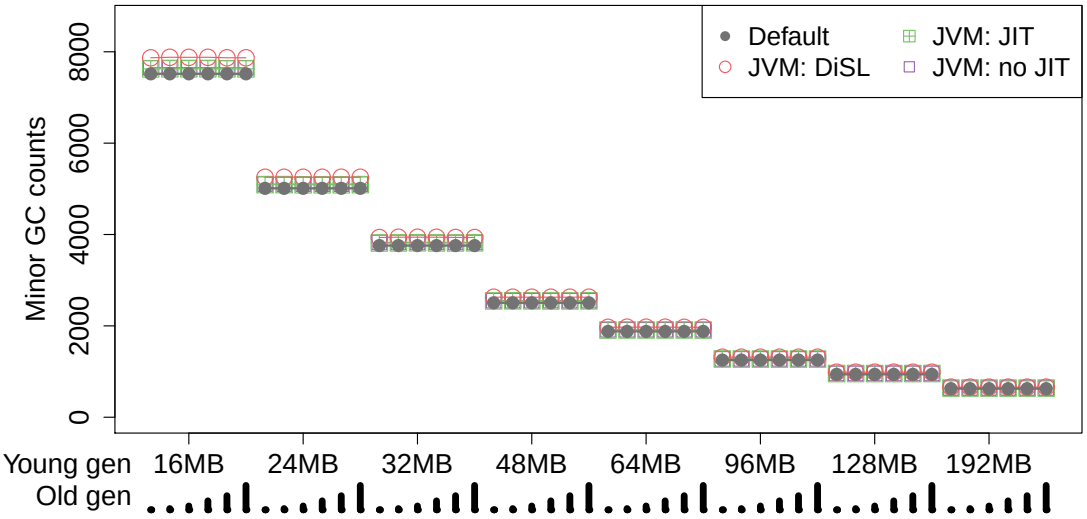


Figure 4.5: Young GC counts – JVM: xalan

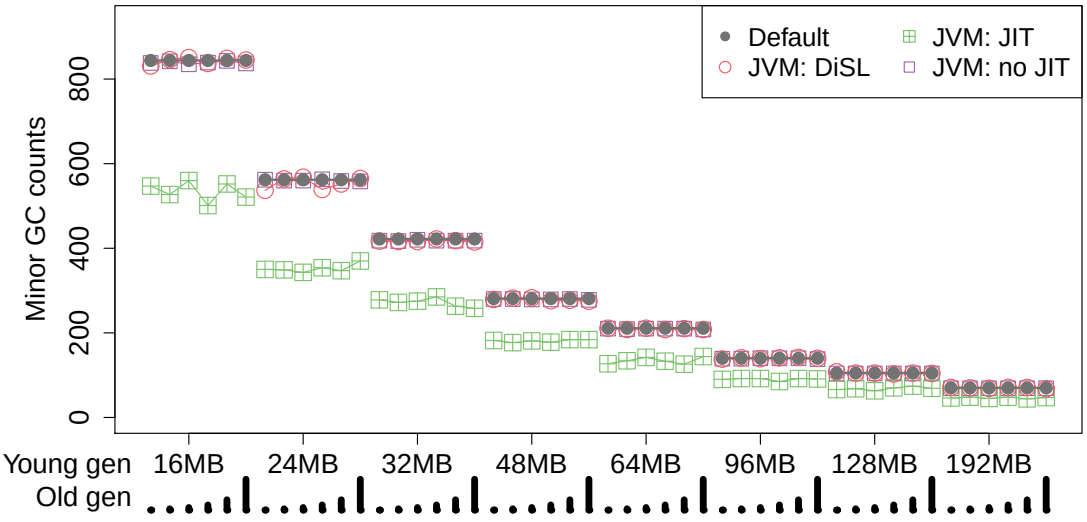


Figure 4.6: Young GC counts – JVM: multifop

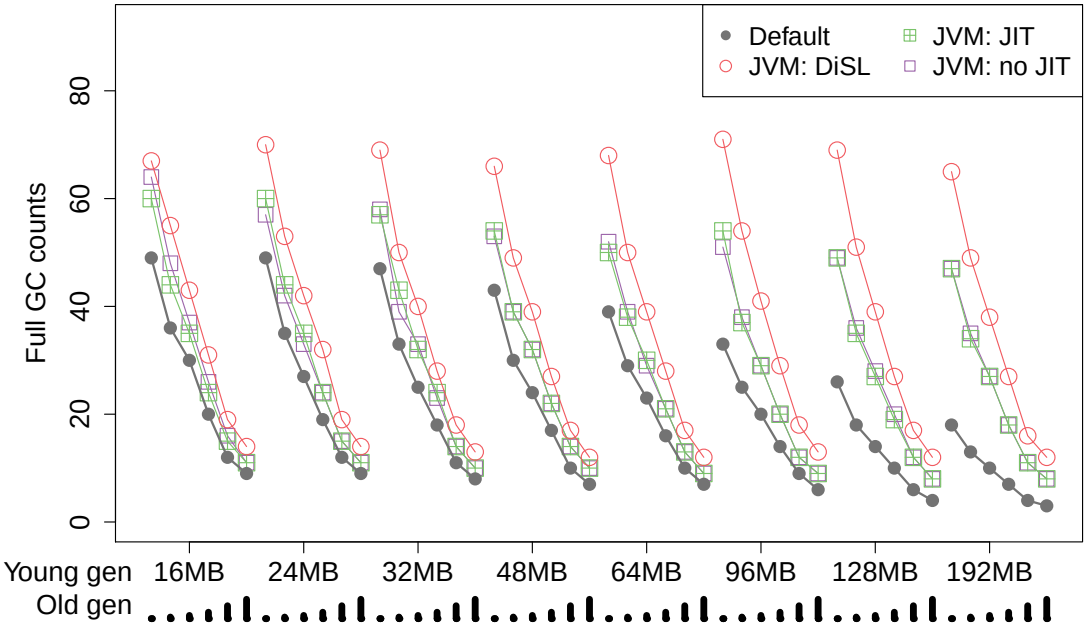


Figure 4.7: Full GC counts – JVM: batik

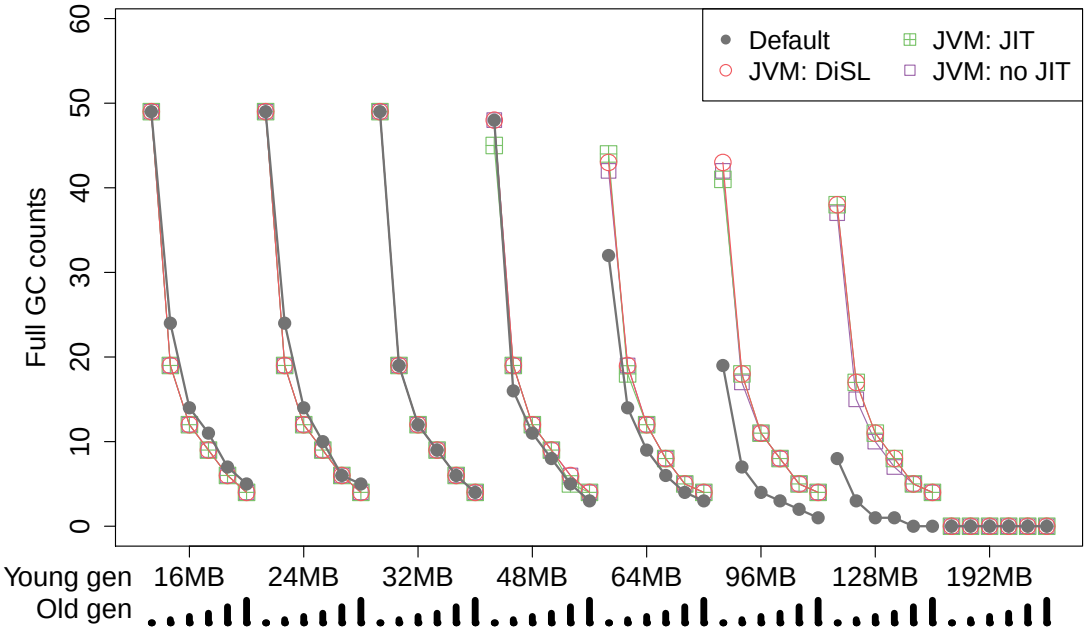


Figure 4.8: Full GC counts – JVM: fop

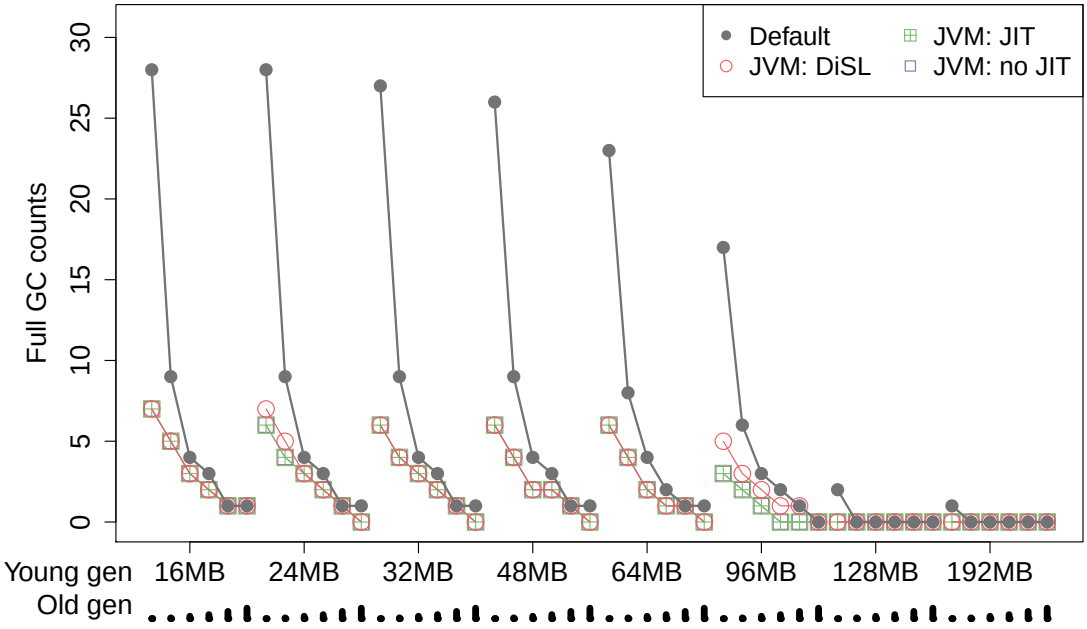


Figure 4.9: Full GC counts – JVM: tomcat

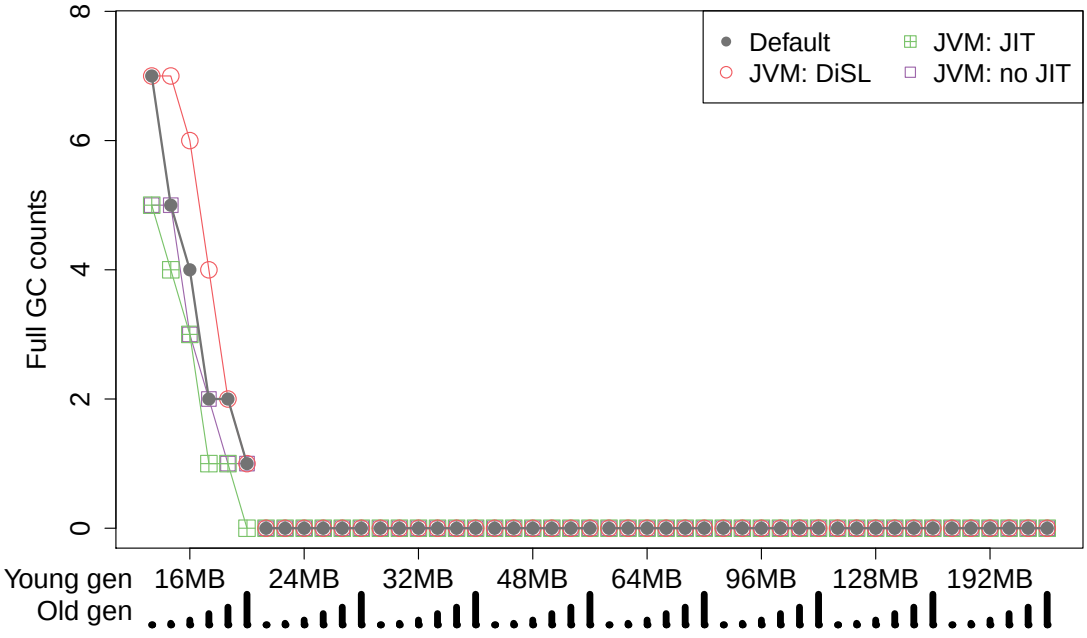


Figure 4.10: Full GC counts – JVM: xalan

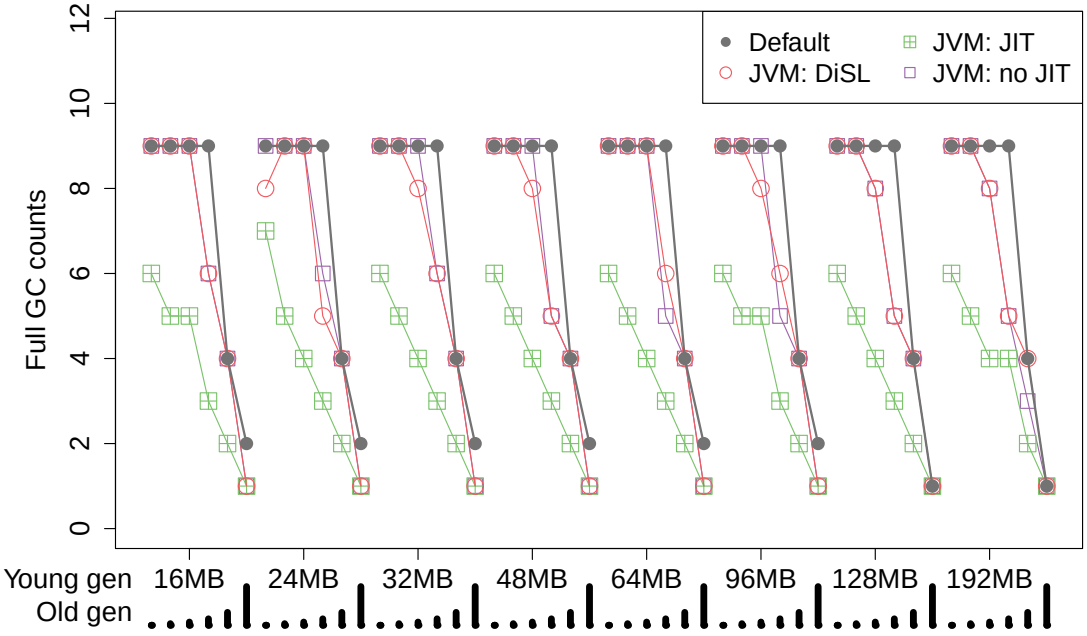


Figure 4.11: Full GC counts – JVM: multifop

4.4 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 Section 4.4.1, we approximate the reference updates information in the input data with a single probability value, leaving only the lifetimes and object sizes. In Section 4.4.2, we experiment with ignoring the reference updates altogether. And finally, in Section 4.4.3, we also replace object lifetimes and sizes with probability distributions.

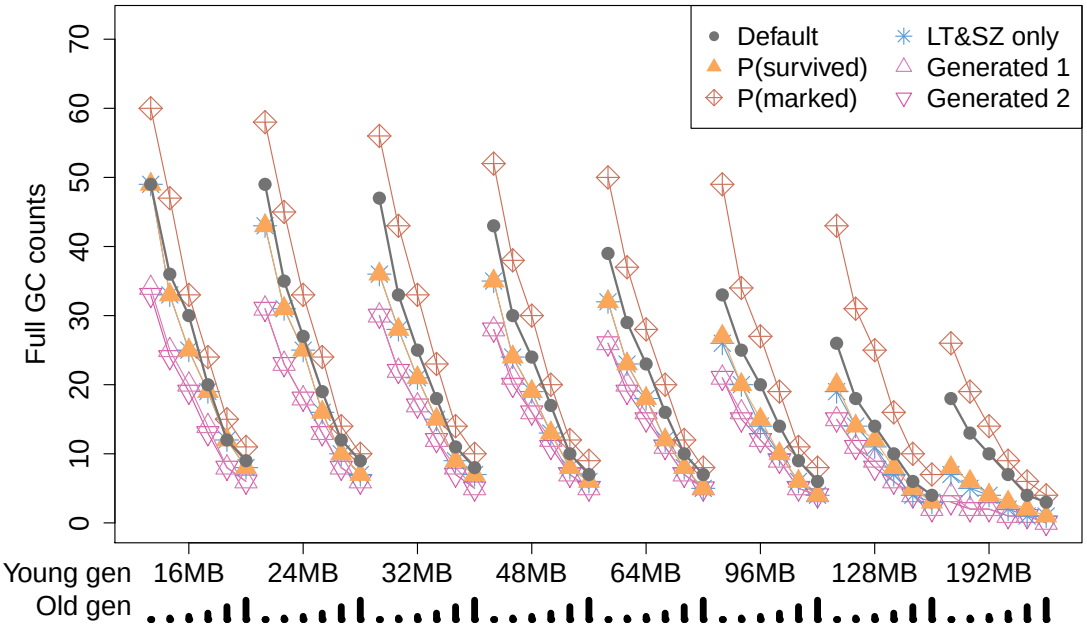


Figure 4.12: Full GC counts – simulators: batik

4.4.1 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

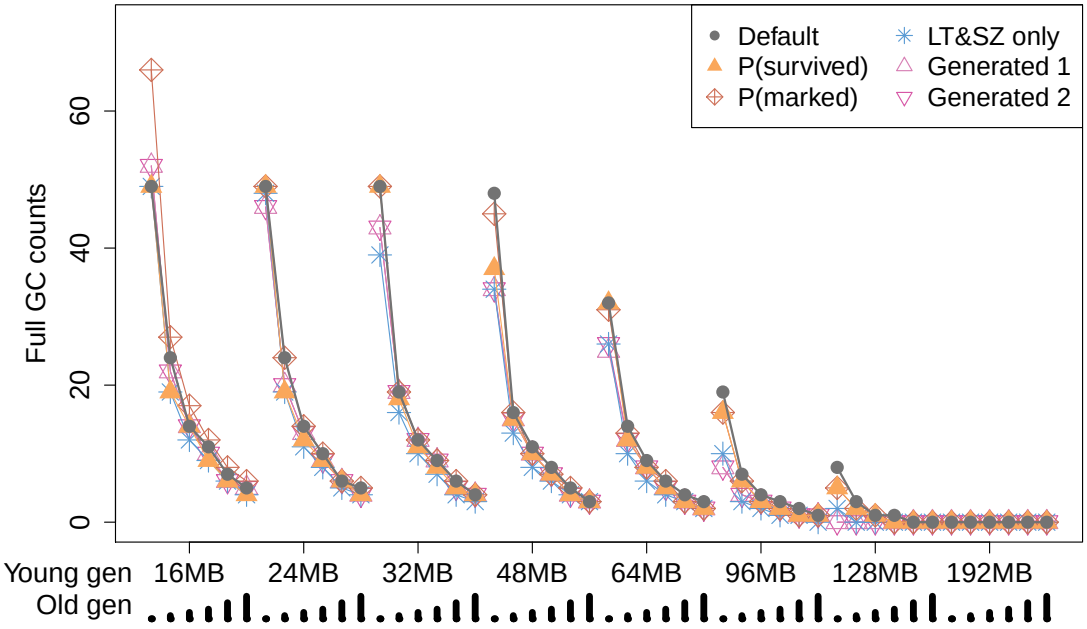


Figure 4.13: Full GC counts – simulators: fop

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 Figure 4.22 and for tomcat in Figure 4.23. 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 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 shown in Figures 4.12, 4.13, 4.14, 4.15 and 4.16 for the full collections and in Figures 4.17, 4.18, 4.19, 4.20 and 4.21 for minor collections. 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.

4.4.2 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 4.12–4.21,

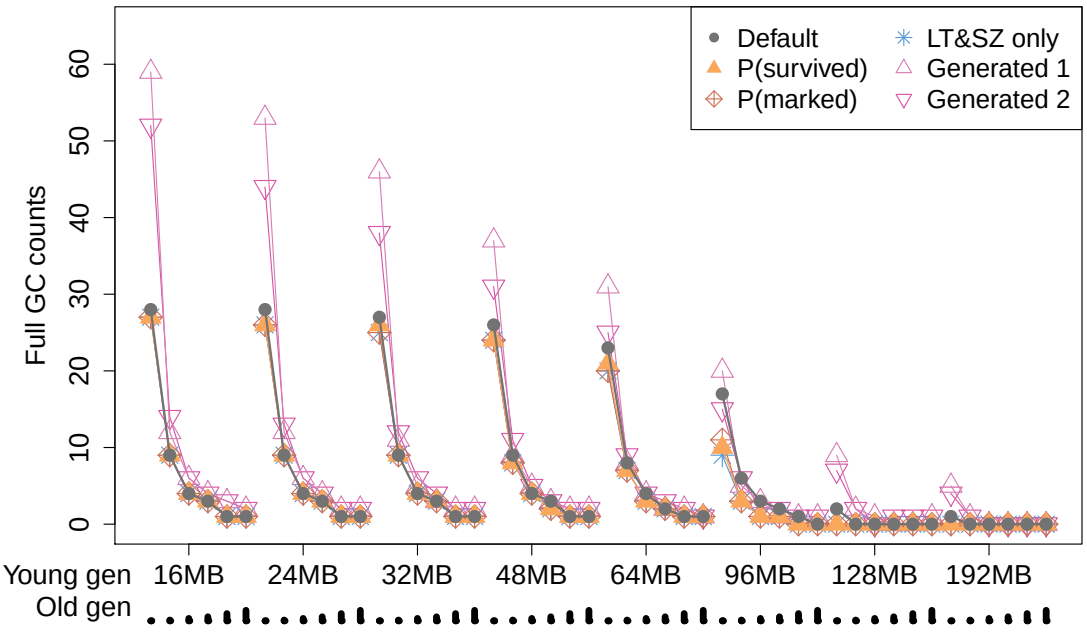


Figure 4.14: Full GC counts – simulators: tomcat

where the results from this experiment are labeled *LT&SZ only*.

4.4.3 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 Section 4.4.1.

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.

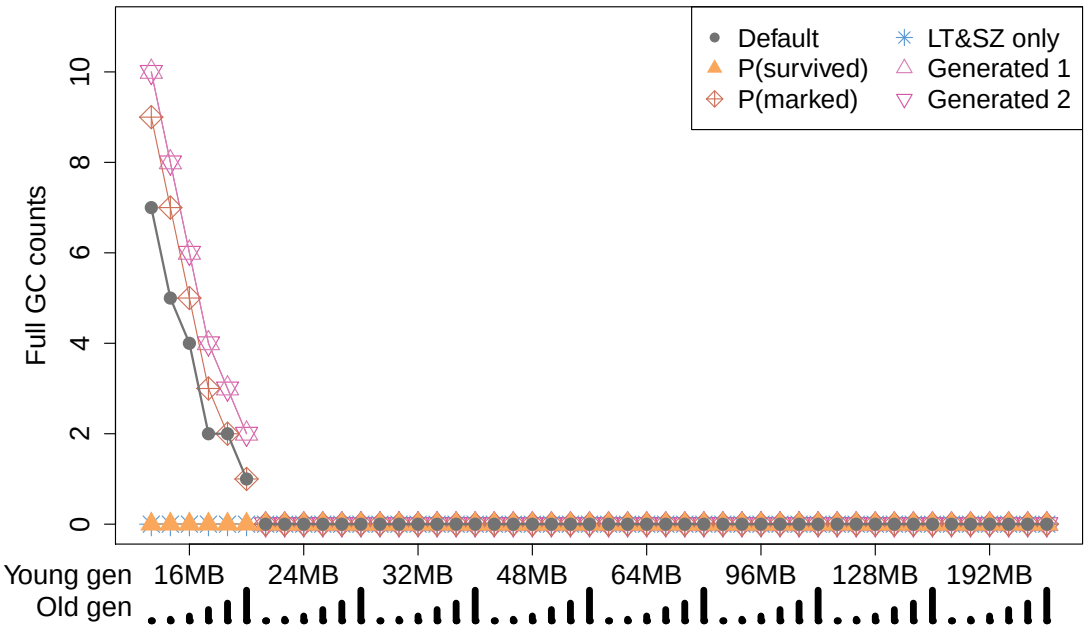


Figure 4.15: Full GC counts – simulators: xalan

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 4.12–4.16 for full collections and in Figures 4.17–4.21 for young collections. We use the legend labels *Generated*

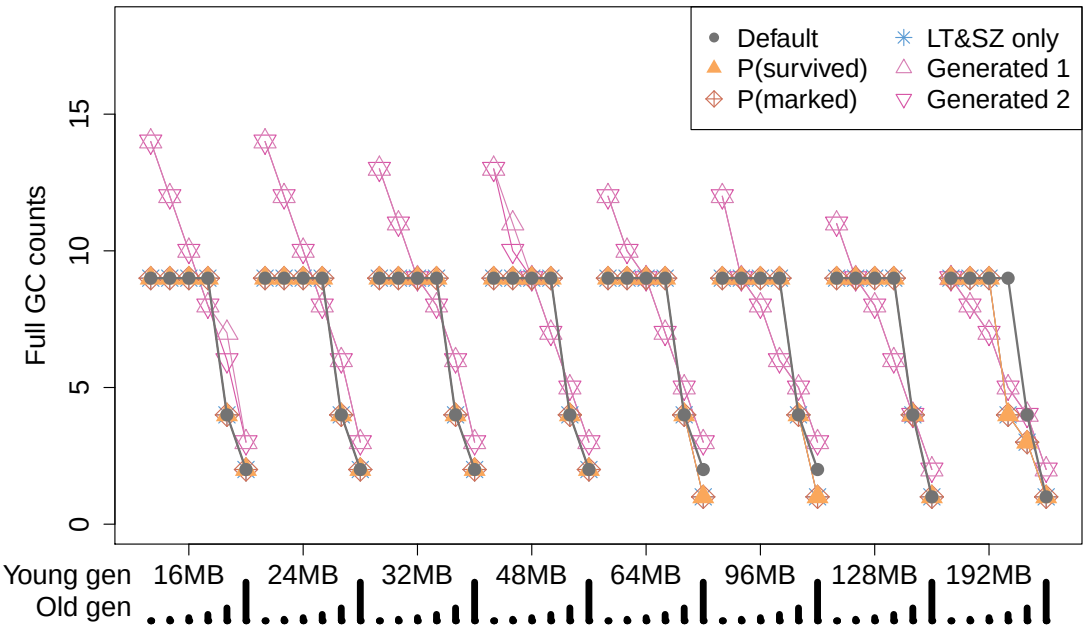


Figure 4.16: Full GC counts – simulators: multifop

1 and *Generated 2* for the data.

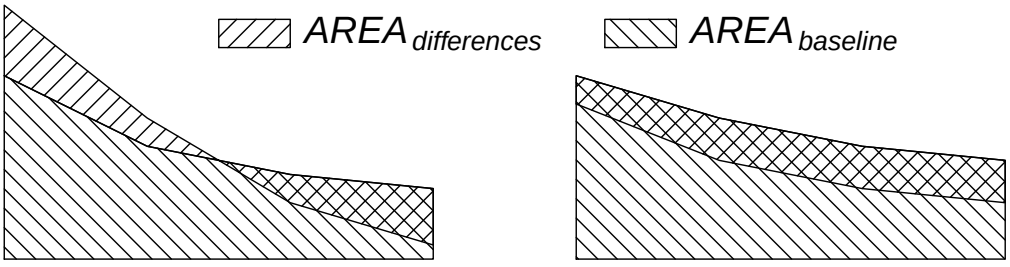
4.4.4 Accuracy Metric

Besides the visual evaluation using the plots in Figures 4.8–4.16, 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:

$$Inaccuracy = \frac{AREA_{differences}}{AREA_{baseline}} \tag{4.3}$$

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.



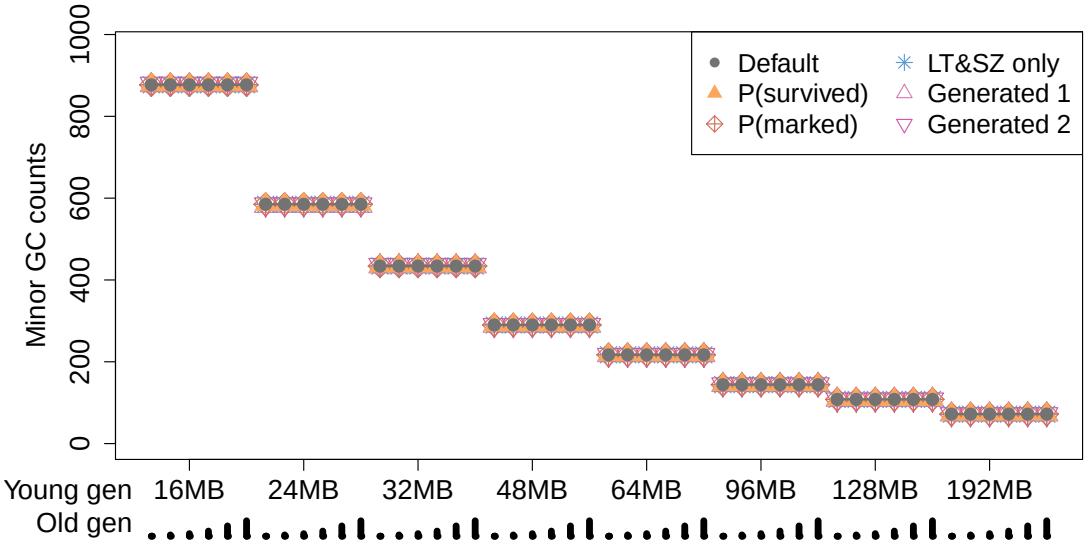


Figure 4.17: Young GC counts – simulators: batik

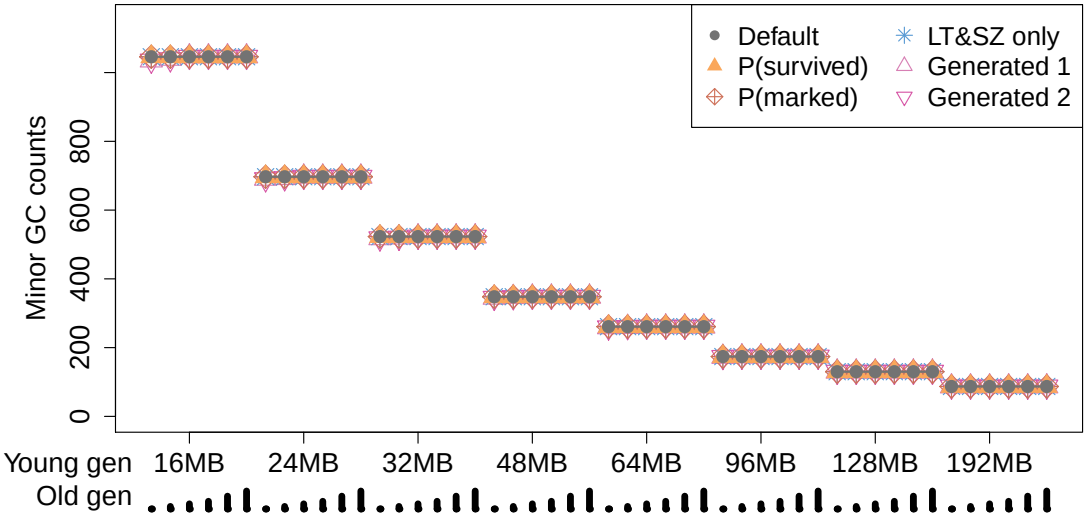


Figure 4.18: Young GC counts – simulators: fop

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 4.3. 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.

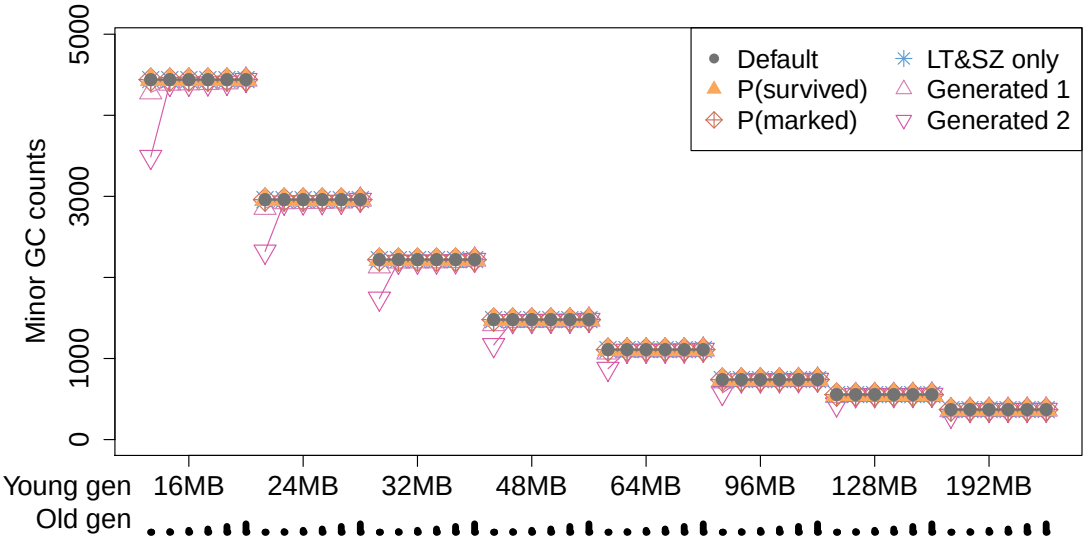


Figure 4.19: Young GC counts – simulators: tomcat

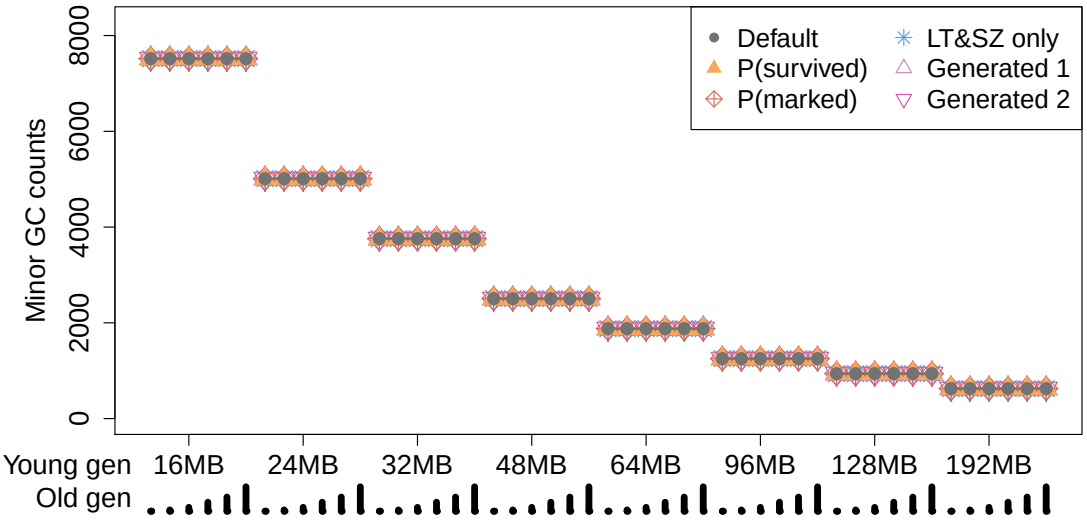


Figure 4.20: Young GC counts – simulators: xalan

4.4.5 Results Discussion

From the results presented on the GC count plots (Figures 4.2–4.21), 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 Section 4.5.

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.

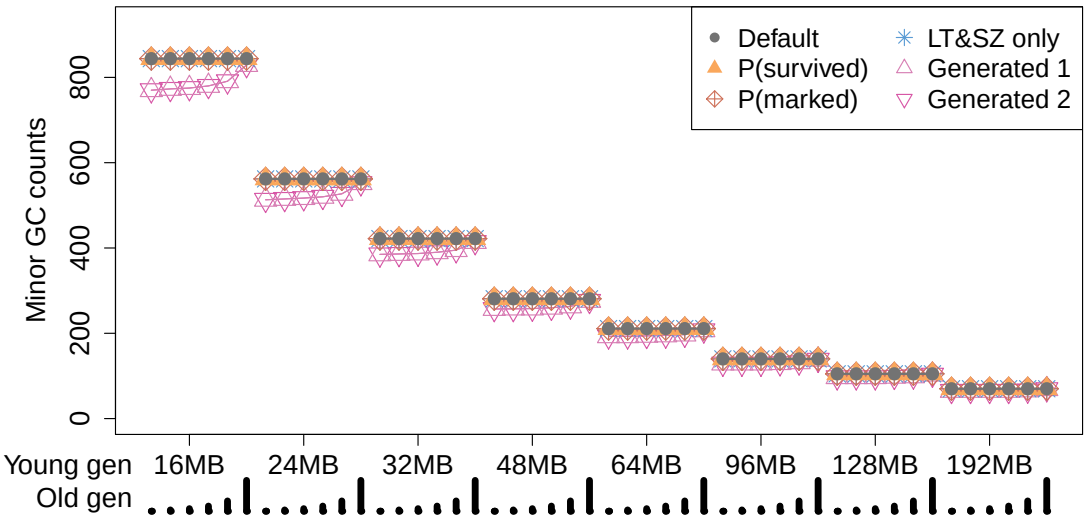


Figure 4.21: Young GC counts – simulators: multifop

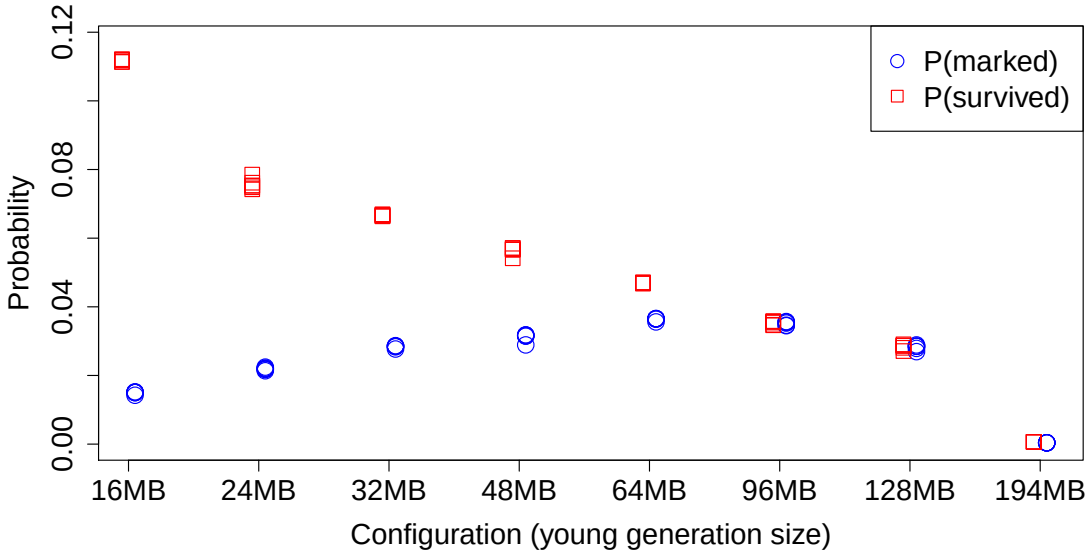


Figure 4.22: Mark probabilities: fop

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).

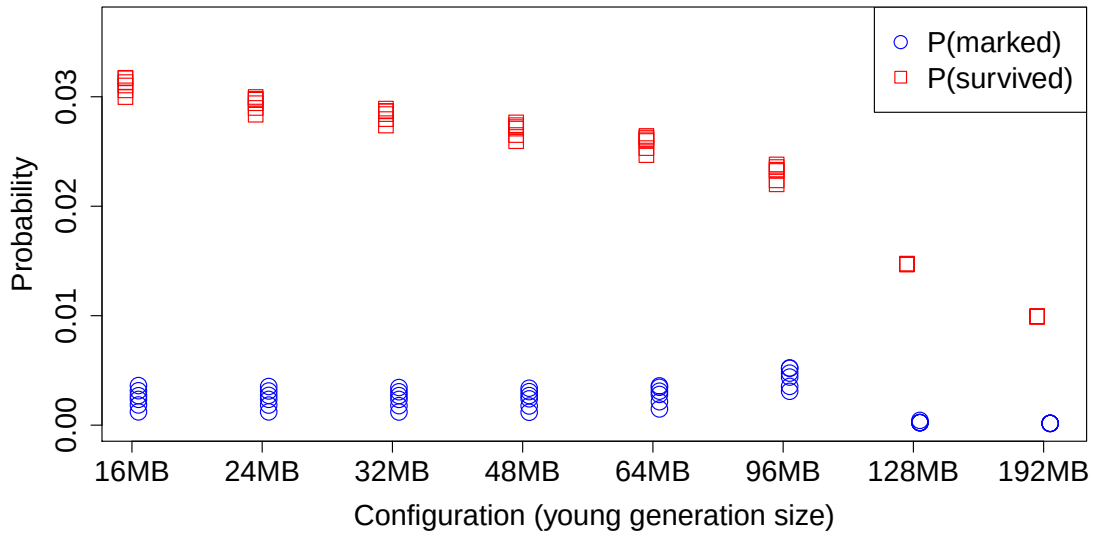


Figure 4.23: Mark probabilities: tomcat

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					

Table 4.3: Inaccuracy for full collections

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 [36]. 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 Figure 4.24, 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.

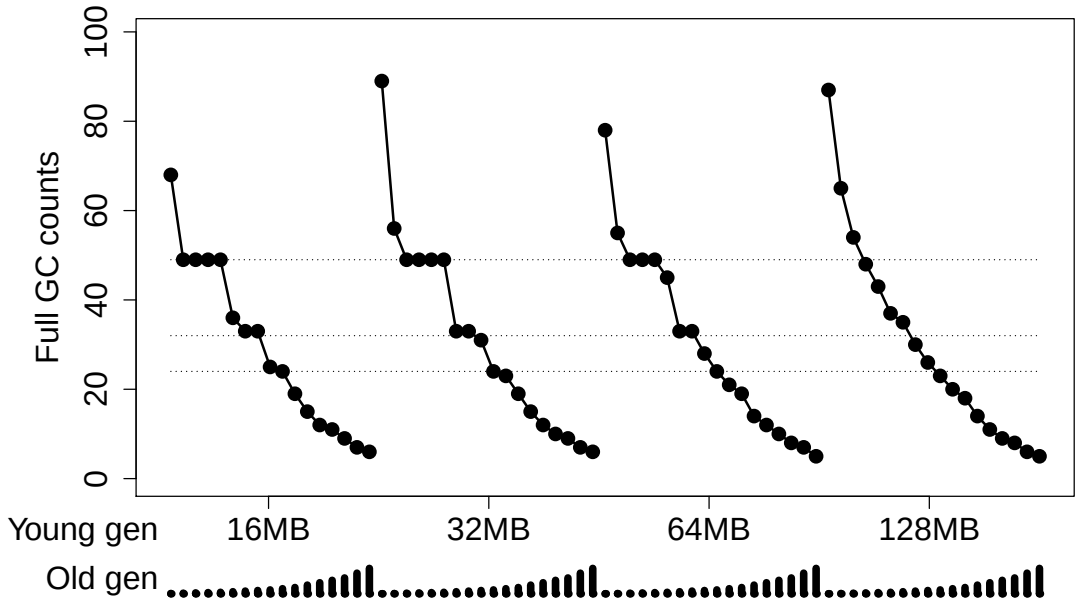


Figure 4.24: Dense configurations: fop

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 4.25 and 4.27 show the first four iterations out of 100 for fop and batik, Figure 4.26 the first four out of 10 for multifop, and Figure 4.28 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 Figure 4.24.

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.

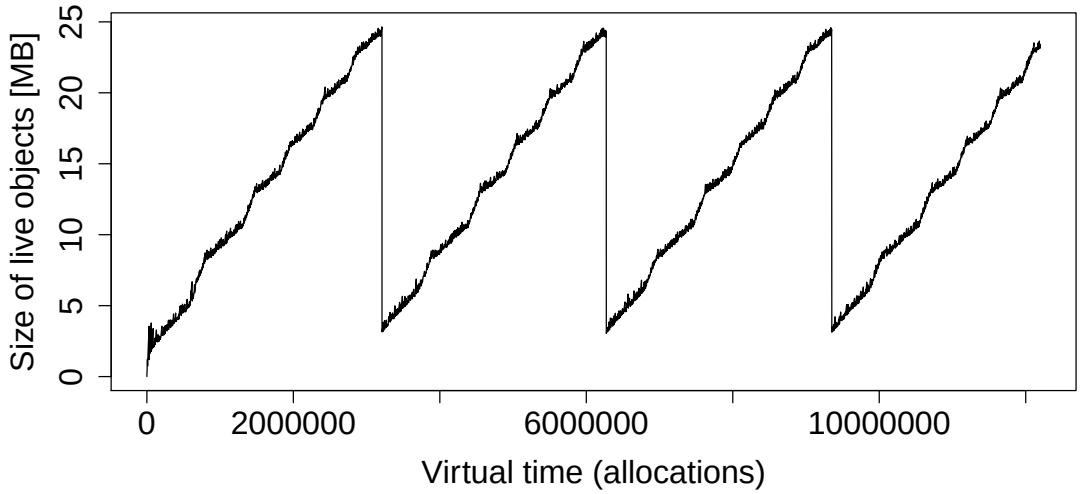


Figure 4.25: Partial live size trace: fop

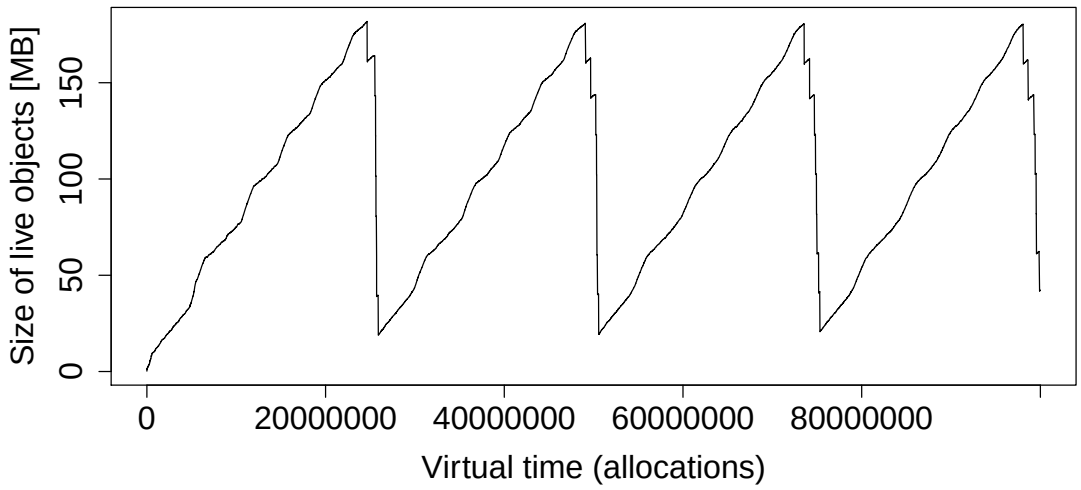


Figure 4.26: Partial live size trace: multifop

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 multifop workload (Figure 4.16), 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.

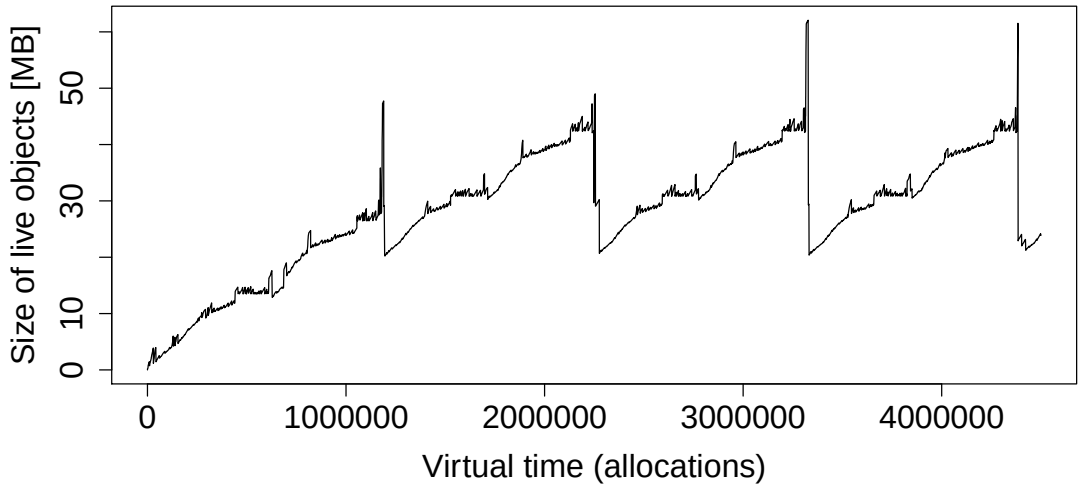


Figure 4.27: Partial live size trace: batik

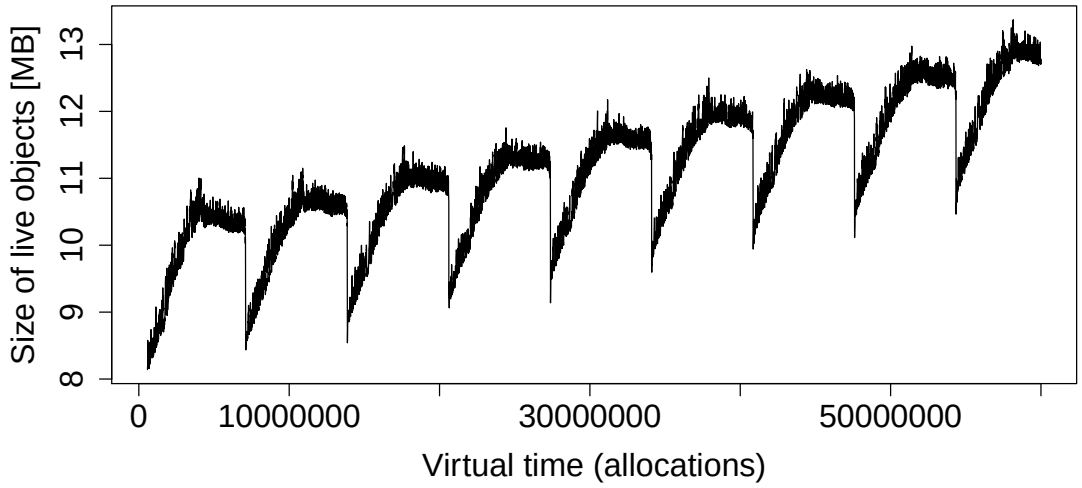


Figure 4.28: Partial live size trace: tomcat

4.5 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 Section 4.3.2). 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. We only report results for changes due to a multiplicative factor, and a random error, as from these we can do the most interesting observations..

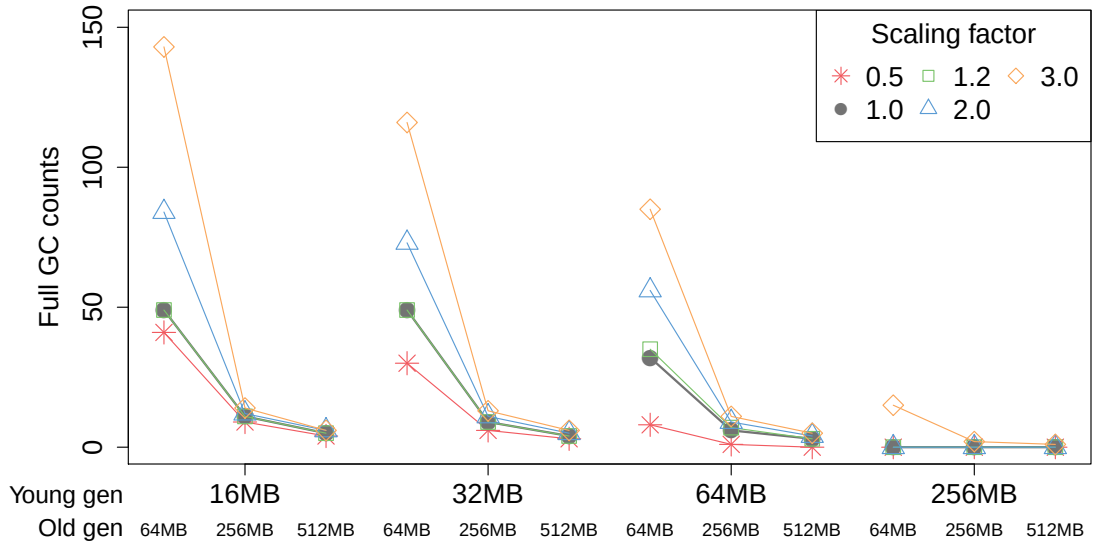


Figure 4.29: Lifetime scaling: fop

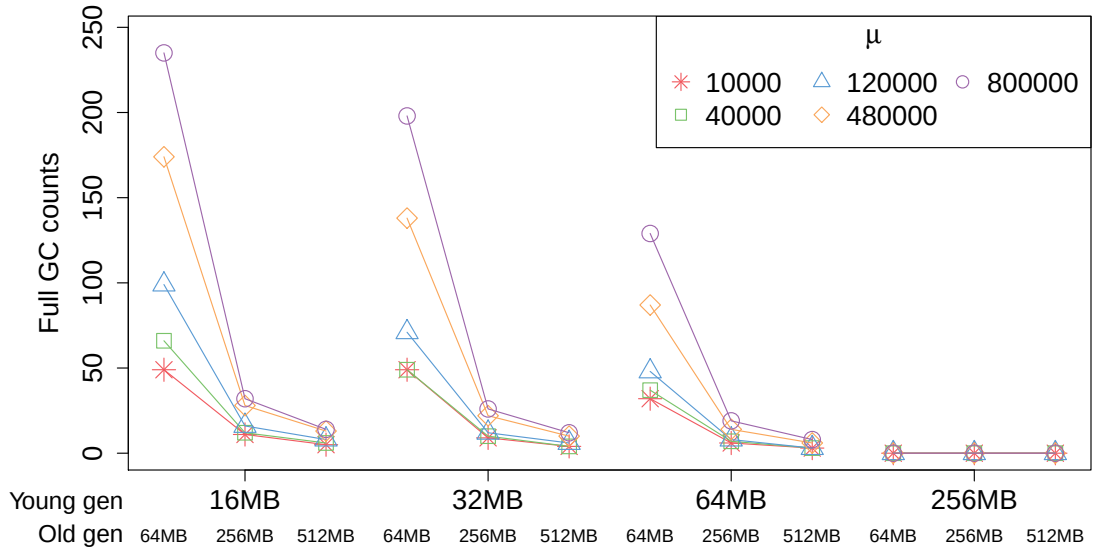


Figure 4.30: Lifetime randomization: fop

4.5.1 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 4.29 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.

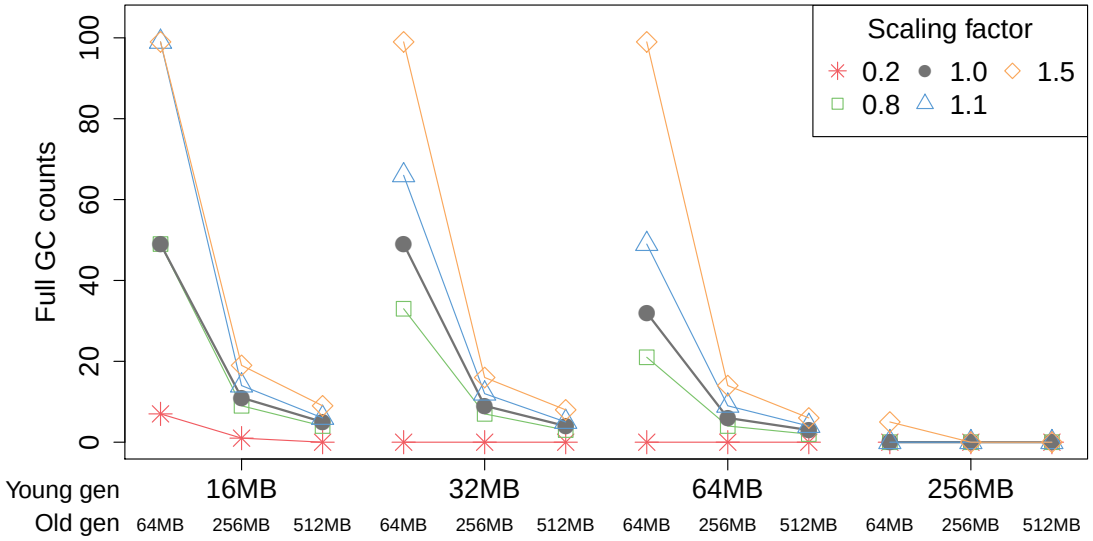


Figure 4.31: Size scaling: fop

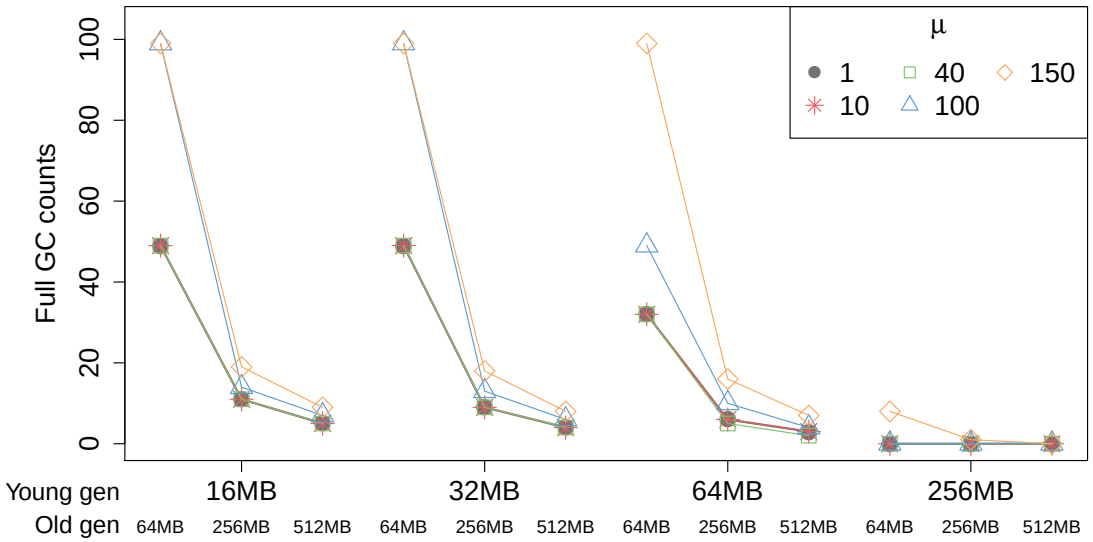


Figure 4.32: Size randomization: fop

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. Section 4.3.2). To include both frequent small deviations and occasional 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 Figure 4.30. 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 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.

4.5.2 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 [2] to reduce memory overhead on 64-bit systems.

Figure 4.31 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 Section 4.4.5—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 Figure 4.32. They again confirm the clustering effect and show that it is not sensitive to small changes in object size.

4.6 Summary of Chapter 4

Motivated by the need to understand garbage collection behavior from the application developer perspective, and some motivating results from one-generation GC, in this chapter we use 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 [63, 70].

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 [36] 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 [65]. 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 [17]. 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.

Complete tools and results are available on-line at http://d3s.mff.cuni.cz/papers/gc-modeling-icpe.

Chapter 5

Estimating Effects of Code Additions on GC Performance

In the previous chapter we have shown it is difficult to have a precise garbage collection performance model—even with extremely large input and detailed simulation the prediction accuracy is not good. This leads us to the question if we can do reasonably precise prediction in constrained situations. Here we explore the settings similar to those of the Q-ImPrESS project [64] in which our group participated. One of the project's important parts was about performance prediction for evolving software.

The scenario of interest here is that we have some non-trivial software which is well tuned and is executing in well optimized environment. Now, we need to implement additional functionality into the software. We know the code location and that the location is frequently executed. The developer has a rough idea how such code addition would look like, in form of a code skeleton, perhaps. However, it is time consuming to fully incorporate the new feature just to realize the system will be too slow afterwards so he wants to find an evidence to show the code addition will not break the application performance.

In such case, the developer can use simplified microbenchmarks in isolation to find out how long single addition instance will execute. He can prepare simple microbenchmarks, preferably using frameworks like jmh [62, 67] that partially alleviate the developer from the difficulties caused by the platform, in our case Java. The jmh framework forces the Java Virtual Machine to execute the code in a manner that resembles the run within bigger, fully JITed application and it can measure the time needed to execute the code of interest. The developer can run the microbenchmarks to get estimation of how long single addition instance will execute and when he knows how often it will be executed in target system he can get additional runtime estimate by simple multiplication. It is clear this is overly simplified, but we ask here if the developer can make too big mistake by relying on microbenchmarks only with regard to garbage collection. The similar scenario is the situation, when the developer is deciding which of several options he has to implement, will at the end have the optimal performance.

Unfortunately, such microbenchmark measurements only take execution time into account and they are unable to grasp effects of mechanisms like garbage collection or cache sharing. In execution environments with managed heap, like

Java, the cost of allocating memory is not necessarily paid in the execution time in the workload iteration, but this is paid later in garbage collection cycle.

We start this chapter with an experiment showing if garbage collection can cause invalidation of microbenchmark measurements in Section 5.1. We design a model to estimate impact of code additions on collector performance, starting by reviewing basic GC features our method relies on in Section 5.2 and we describe the model itself in Section 5.3. We evaluate the model and discuss our findings in Section 5.4.

We published the model, evaluation and discussion in:
[50] P. Libič, L. Bulej, V. Horký, and P. Tůma. Estimating the impact of code additions on garbage collection overhead. In *Proceedings of the 12th European Conference on Computer Performance Engineering*, EPEW'15, Berlin, Heidelberg, 2015. Springer-Verlag

5.1 Motivating Experiment

We start with an experiment to see if it is possible that the performance is different when executing in microbenchmark and in bigger application due to garbage collection, and if it can turn the performance decision around, or how big the difference in allocated memory has to be to expose the different garbage collection cost. Our piece of code to experiment with attempts to have two tunable settings, one changing the execution time and the second setting changing the amount of the memory allocated by the code. Ideally, these two setting should be independent and the code should always exercise the same code path.

We created a small piece of code where we can set how much memory it allocates by setting size of an array and how long it executes by setting number of operations—array reads in this case. Unfortunately, with Java it is very difficult to achieve the desired independence, because Java always clears memory allocated for arrays and this takes longer time for bigger arrays. The independence can be achieved only by using different object graphs for different settings, but then the code will execute different paths.

The lack of independence is a slight drawback of the code but it does not influence the results in significant manner. We also took precaution for compiler not to optimize out our work loop and the data allocation. The resulting code is shown in Listing 5.1.

```
public class Addition {
  public static volatile int array_reads = 2048;
  public static volatile int array_length = 2048;
  /* must be less or equal to array_length */
  public static volatile int walk_length = 2048;
  /* deterministic seed */
  static Random rnd = new Random(1);
  public static volatile long sum = 0;

  public static void work() {
    int[] data = new int[array_length];
    data[array_length - 1] = rnd.nextInt();
```

```
    /* Initializing array */
    for (int i = 0; i < walk_length; i++){
      data[i] = rnd.nextInt();
    }
    /* Work */
    for (int step = 0; step < array_reads; step++) {
      sum += data[rnd.nextInt(walk_length)];
    }
  }
}
```

Listing 5.1: Addition code with variable allocated memory and amount of work

We executed the code within the jmh framework[1] running on Dell PowerEdge 1955 system[2] and Oracle HotSpot 1.8.0_11 JVM[3] One set of measurements was done with fixed number of operations—array reads to 2048 reads and varying the array length. Results are in Table 5.1a. In the second set we fixed the array length to 2048 ints and varied the number of array reads, the results are in Table 5.1b.

Array size	Iteration time (ns)	Error (ns)
2048	144163	13.9
2304	144499	14.5
2560	**145128**	**15.6**
3072	146087	16.9
4096	149096	19.7
6144	*152370*	*22.9*

(a) Changing array length, 2048 array reads

Array reads	Iteration time (ns)	Error (ns)
2048	144096	14.0
2176	149814	14.4
2304	***155463***	***14.3***
2432	161287	15.4
2560	166860	14.8
2688	172607	15.7

(b) Changing array reads, array length is 2048 ints

Table 5.1: Addition microbenchmark results

In the next step, we included the code fragment (from Listing 5.1) into larger benchmark, our dbart code simulating student database. We modified operation that records new courses for students, this place is executed 5935084 times in total. Then we executed the larger benchmark with the same settings of the additional fragment as before: the results with fixed number of array reads (2048)

[1] We used following jmh command line options: -tu ns -i10 -r 60 causing the framework to use warmup with 20 iterations, 1 second each, measurement with 10 iterations, 60 seconds each and timeout 10 minutes per iteration, sampling time mode

[2] 24 GB RAM, two Intel Xeon E5345 (Clovertown) chips, 8 processors.

[3] Java(TM) SE Runtime (1.8.0_11-b12), and Java HotSpot^{TM}64-Bit VM (25.11-b03).

are in the Table 5.2a and the results with fixed array length (2048 ints) and varying array reads are in Table 5.2b. These measurements were executed on the same machine and VM as the jmh benchmarks, each configuration executed 6 times and we report average execution time and standard deviation. We used following command line settings: -XX:ParallelGCThreads=1 -Xmx640m -Xms640m -XX:NewSize=192m -XX:MaxNewSize=192m -XX:-UsePSAdaptiveSurvivorSizePolicy -XX:SurvivorRatio=1 -XX:MaxTenuringThreshold=4 -XX:InitialTenuringThreshold=4 to fix the heap settings and space sizes at configuration that is generating some premature promotions, but not too often and the heap size that is only twice the size of live data to have more visible GC overhead. The original unmodified code executes in 4481 seconds and the standard deviation of 9.4 and total time spent doing the garbage collection is 2978 seconds with standard deviation 6.9.

Addition's array length	Runtime (s)	sd	GC time (s)	sd
2048	5549	4.8	3195	7.1
2304	5623	5.6	3263	3.9
2560	**5638**	**6.9**	**3278**	**6.5**
3072	5670	4.7	3308	5.6
4096	5728	5.2	3348	3.8
6144	*5919*	*7.8*	*3519*	*6.6*

(a) Changing array length, 2048 array reads

Addition's array reads	Runtime (s)	sd	GC time (s)	sd
2048	5548	5.5	3190	6.3
2176	5580	1.8	3192	3.0
2304	***5614***	***8.3***	***3191***	***9.0***
2432	5652	5.6	3193	4.7
2560	5678	7.3	3188	7.4
2688	5718	4.7	3192	5.7

(b) Changing array reads, array length is 2048 ints

Table 5.2: dbart execution times with included additions

Now, we can imagine the developer has three options to choose from, that behave similar to our artificial additions with these settings: (1) array size 2560 and 2048 array reads, with **bold font** in Tables 5.1 and 5.2, (2) array size 6144 and 2048 array reads, with *italics* and (3) array size 2048 and 2304 array reads, with ***bold italics***. With the microbenchmark results from Table 5.1 the developer would evaluate as the option with the best performance one with characteristics (1) followed by (2) and (3) as the worst (145 µs vs. 152 µs vs. 155 µs). However, in the full program case the order is different: the best is the slowest option from microbenchmarks—(3) followed by (1) and (2) has the worst performance (5614 s vs. 5634 s vs. 5919 s). Especially choice of the option (2) would cause significantly suboptimal performance.

In light of these results, in this chapter we define and evaluate a model that estimates collection behavior after adding code fragments with additional alloca-

tions into original code, requiring the developer only to identify code location and amount of data allocated in the addition. We make the estimate with reasonable input collection and model calculation time in mind, which is in contrast with the models presented in Chapter 4, where we collect enormous amount of data and then reading this data alone takes longer than the execution of the original code. This limits the possible additional code fragment the approach is suitable for along with requiring the original application is in steady state concerning its allocation behavior. In these limited settings, we propose a model in this chapter that only requires the GC log and information on free space, which are both commonly available in today's virtual machines.

[50]

5.2 Garbage Collection Essentials

Our overhead estimation method has been developed in the context of the default (parallel throughput) garbage collector (described in Section 2.4) of the HotSpot virtual machine. In this section, we list the essential GC features we rely on. Possibly, our model can be applied also to other collectors with similar features.

Architecture. The collector architecture is generational. It separates objects into *young generation* and *tenured generation* and uses two configurable collectors, one to collect the young generation, one to collect both generations.

The default young generation collector (sometimes called *parallel scavenging*) is a copying collector. The memory allocated for the young generation is separated into one *eden* area and two *survivor* areas. New objects are allocated (mostly) in eden, each young generation collection copies reachable objects from eden and one survivor into the other survivor. Objects that survive more young collection count than tenuring threshold, are promoted to the tenured generation. (§1)

The default full collector is a mark-compact collector. The young generation is evacuated into the tenured space on each collection. (§2)

Dimensioning. The generations have default dimensions derived from the memory capacity of the execution platform. Ergonomics (adaptive sizing policy) is used to dynamically scale the generations. This can sometimes lead to performance anomalies (as we demonstrate in Chapter 3), which is why manual sizing is recommended for production deployment and we assume fixed sizes in our model.

A young collection is triggered whenever the eden in the young generation is full. During collection, the tospace survivor may overflow, leading to *premature promotion* of the remaining objects to the tenured generation. These facts can be (§3) used to configure the young generation—it should be big enough to avoid excessive overhead due to frequent collections but small enough to prevent individual collections taking too much time, and the tenuring threshold should be small enough to prevent frequent premature promotions. (§4)

A full collection is triggered whenever the tenured generation is close to full. Some reserve is maintained to prevent promotion failures in young collections, the size of this reserve is derived dynamically as a weighted average of the amount of promoted objects. (§5)

Monitoring. The HotSpot virtual machine provides support for displaying information about heap occupation at GC events. The information is recorded in
(§6) the GC log, whose short example follows:

```
[GC (Allocation Failure) [PSYoungGen: 128960K->65536K(131072K)]
 ↪  536389K->489317K(589824K), 0.5624948 secs]
 ↪  [Times: user=0.57 sys=0.00, real=0.56 secs]
[GC (Allocation Failure) [PSYoungGen: 131072K->65536K(131072K)]
 ↪  554853K->509517K(589824K), 0.5823409 secs]
 ↪  [Times: user=0.59 sys=0.00, real=0.58 secs]
[Full GC (Ergonomics) [PSYoungGen: 65536K->0K(131072K)]
 ↪  [ParOldGen: 443981K->322940K(458752K)]
 ↪  509517K->322940K(589824K),
 ↪  [Metaspace: 2938K->2938K(1056768K)], 6.4256753 secs]
 ↪  [Times: user=6.42 sys=0.00, real=6.43 secs]
[GC (Allocation Failure) [PSYoungGen: 65536K->16064K(131072K)]
 ↪  388476K->339004K(589824K), 0.1873043 secs]
 ↪  [Times: user=0.19 sys=0.00, real=0.19 secs]
[GC (Allocation Failure) [PSYoungGen: 81600K->32672K(131072K)]
 ↪  404540K->355612K(589824K), 0.2772794 secs]
 ↪  [Times: user=0.28 sys=0.00, real=0.28 secs]
```

For each young collection (first line above), we have the collection reason, the size of the young generation before and after the collection, the size of the entire heap before and after collection, and the collection time. For each full collection (second line above), we additionally have the size of the tenured generation before and after collection.

Outside the garbage collection events, an application can also use a standard interface to query information on the amount of free memory in the virtual ma-
(§7) chine. Although the exact meaning of the provided values is not documented, subtracting consecutive values provides an estimate on the amount of object allocations.

5.3 Modeling Garbage Collection Overhead

Besides the obvious time spent traversing and collecting heap content, GC may impose overhead for example by trashing memory cache content, adding barriers to memory access operations, enforcing particular object layout or reference structure, and so on. Although the overhead can be measured by comprehensive experiments [35], the interactions involved are too many to be captured in a white box model of reasonable complexity.

Rather than modeling GC overhead for an entire application—a task that requires detailed input on application allocation behavior even for partial tasks such as modeling GC frequency (as we learned in Sections 4.3 and 4.4)—we focus on modeling the impact of certain application modifications on the total GC
(§8) time. We consider modifications that add allocations of short-lived objects into particular application locations—in practice, these are for example minor code patches, insertion or activation of features such as logging, modifications that change optimization decisions and turn stack allocations into short-lived heap allocations, and so on.

In contrast, we do not consider modifications that allocate significant amounts of long-lived objects. This is relatively strong assumption, we explain why it is

necessary in Section 5.4.6 after we describe the model. We also assume applications that have a relatively stable allocation behavior, rather than passing through phases whose allocation behavior varies significantly. We discuss these assumptions in more detail together with the model description—we believe they represent reasonably minimal constraints for a model that does not require expensive inputs. (§9)

We decompose the problem of estimating the total GC time into estimating the time of each collection and estimating the collection frequency. To estimate the time, we use gray box modeling with assumptions about algorithmic complexity of the GC algorithms involved.

For estimating frequency, we look at the application allocation behavior.

In general, GC is run to keep up with the application allocation rate. The collection frequency can therefore be expressed simply as a ratio of the allocation rate and the free heap size—for example, an application that allocates $100\,\mathrm{MB\,s^{-1}}$ in a heap that has 1 GB free after each collection will need to execute GC every $1\,\mathrm{GB}/100\,\mathrm{MB\,s^{-1}} = 10\,\mathrm{s}$. Unfortunately, neither the allocation rate nor the free heap size are constants, hence the example computation can rarely be applied directly.

The detailed allocation behavior of an application can be expressed in the form of an object lifetime trace, where each allocation and deallocation event is recorded on a timeline [36, 65]. Such trace can be used to deduce both the allocation rate and the free heap size, providing a garbage collection frequency estimate (as presented in Section 4.3), but the approach is relatively expensive in terms of both input collection and model computation.

Our model therefore uses only partial information on the allocation behavior of an application, that can be reconstructed from the information provided in the GC log (§6).

In the following derivations, we use verbose variable and function names to make the formulas more readable. For example, the heap dimensions are denoted as *max.size.eden*, *max.size.survivor*, *max.size.tenured*. The information from the GC log is *log.young.before* and *log.young.after* for the young generation occupation, *log.heap.before* and *log.heap.after* for whole heap occupation. The current occupation of the heap is denoted *in.eden*, *in.survivor*, *in.tenured*, obviously $in.young = in.eden + in.survivor$ and so on.

Some symbols refer to information concerning a particular garbage collection. When presented without additional specification, the symbols refer to the current collection in the discussion context. Symbols with subscript refer to a particular collection index or rank. We index young collections following a full collection starting from 1, and also define collection rank r as the collection index i capped one collection above the tenuring threshold, $r = \min(i, tenuring.threshold + 1)$.

5.3.1 Reconstructing Allocation Behavior

Central to our model is the construction of a function that approximates how objects survive young collections. The function is directly related to object lifetime—only objects whose lifetimes exceed that of the particular young collection survive, objects whose lifetimes are shorter are collected.

We use the information from the GC log, specifically the sizes of the young generation and the entire heap before and after each young collection (§6). Obviously, we also have $log.tenured.before = log.heap.before - log.young.before$ and $log.tenured.after = log.heap.after - log.young.after$. [4]

After a full collection, the young generation is empty (§2). The first young collection following a full collection (rank 1) is triggered when the eden is full and both survivors are empty, we therefore have $log.young.before_1 = max.size.eden$ as the amount of objects allocated during one young collection period. Denoted as $surviving_1$, $log.young.after_1$ is the amount of surviving objects allocated during one young collection period.

To further clarify where $surviving_1$ and following come from we complement the textual description with several illustrations. The first one depicts situation right after the first young garbage collection that follows a full collection (rank 1), in Figure 5.1. The boxes depict state of the eden space and survivor space (there are two spaces in reality, but one is always empty after garbage collection). The areas with the same color represent space occupied by objects that were allocated before young collection where this hatching/shading/color fills the entire eden space. Arrows depict where the objects are copied. In reality, the objects allocated between different collections will be mixed in survivor, here we just illustrate the sizes they occupy.

The second young collection following a full collection (rank 2) is again triggered when the eden is full, we therefore have $log.young.before_2 = max.size.eden + log.young.after_1$. The amount of surviving objects allocated during two young collection periods, $surviving_2 = log.young.after_2$, consists of objects that have survived two young collections (allocated before the first young collection) and objects that have survived one young collection (allocated after the first young collection).

The situation after rank 2 collection is illustrated in Figure 5.2. In the rank 2 collection the data kept in survivor after rank 1 collection—$surviving_1$—and still alive will be in the second survivor space, but probably not all of them because some objects can be dead already. Plus the survivor will contain objects surviving from eden space, together occupying $surviving_2$ space. Here we depicted the amount surviving rank 2 collection from eden being the same as $surviving_1$—the amount surviving rank 1 collection. This is ideal case with completely stable allocation behavior (§9). It is unlikely to happen in real applications but it is needed for the model.

We can proceed inductively for as long as no objects are promoted. When young collection with index i promotes some objects, the savings in the young generation will not match the savings in the entire heap:

$$log.young.before_i - log.young.after_i \neq log.heap.before_i - log.heap.after_i$$

When this happens, it no longer holds that $surviving_i = log.young.after_i$, because $log.young.after_i$ does not include the promoted objects. We can, however,

[4] Some configurations of HotSpot can display object lifetime distribution at GC events, however, that feature is not available in the default collector and not sufficiently complete in other collectors.

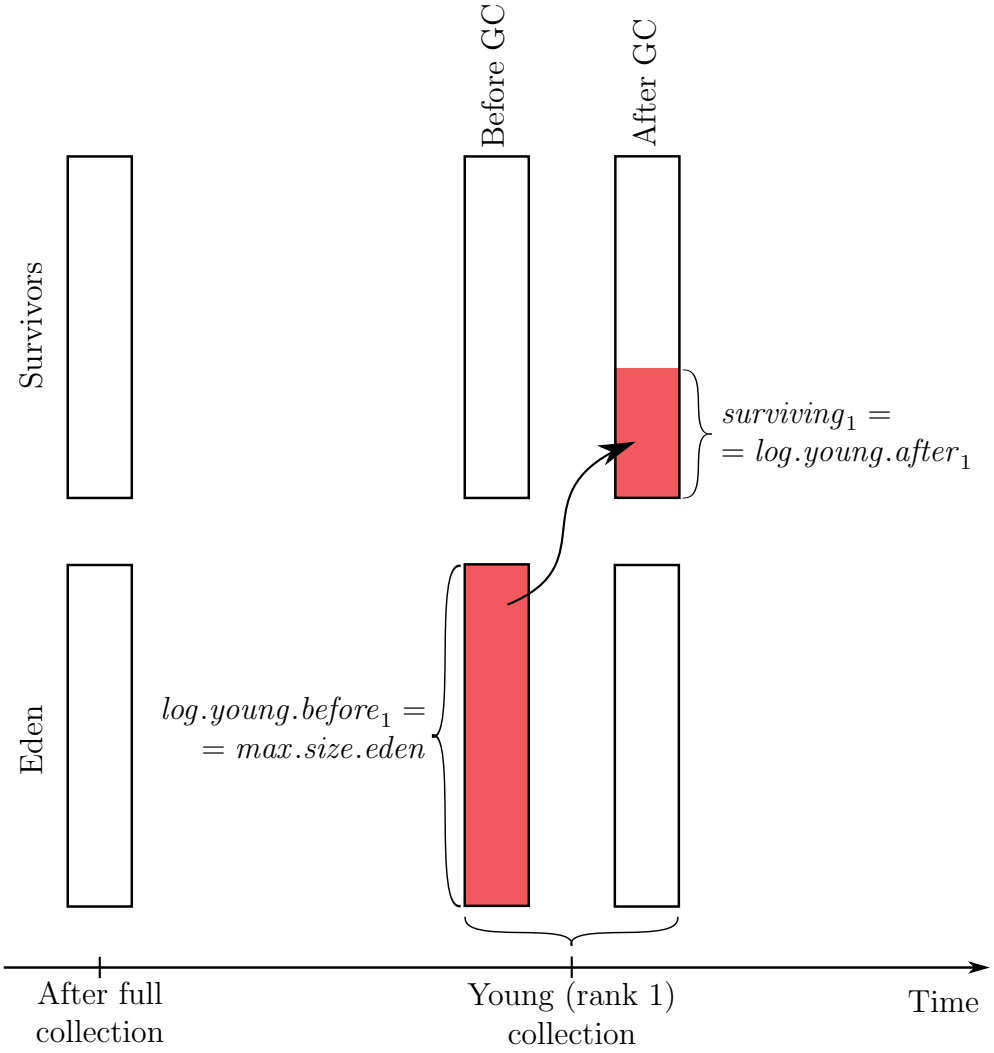


Figure 5.1: $surviving_1$ illustration

still salvage the computation of $surviving_i$:

$$surviving_i = log.heap.after_i - log.heap.before_i + log.young.before_i$$

After a promotion in young collection with index i, we no longer have enough information to compute the amount of surviving objects $surviving_j$ for $j > i$, simply because the liveness of promoted objects is only examined during a full collection. However, if the promotion is due to objects reaching the tenuring threshold (§1), we can still use the young collections with index $j > i$ to compute additional estimates for $surviving_i$:

$$surviving_i = log.heap.after_j - log.heap.before_j + log.young.before_j$$

Figure 5.3 illustrates the situation after the young collection with rank 4 for the case $tenuring.threshold = 3$. The $surviving_4$ variable includes also the size of objects promoted in rank 4 collection which is depicted in the illustration as being promoted into tenured space.

Because every full collection empties the young generation, we can repeat the same computations as many times as there are full collections, obtaining multiple

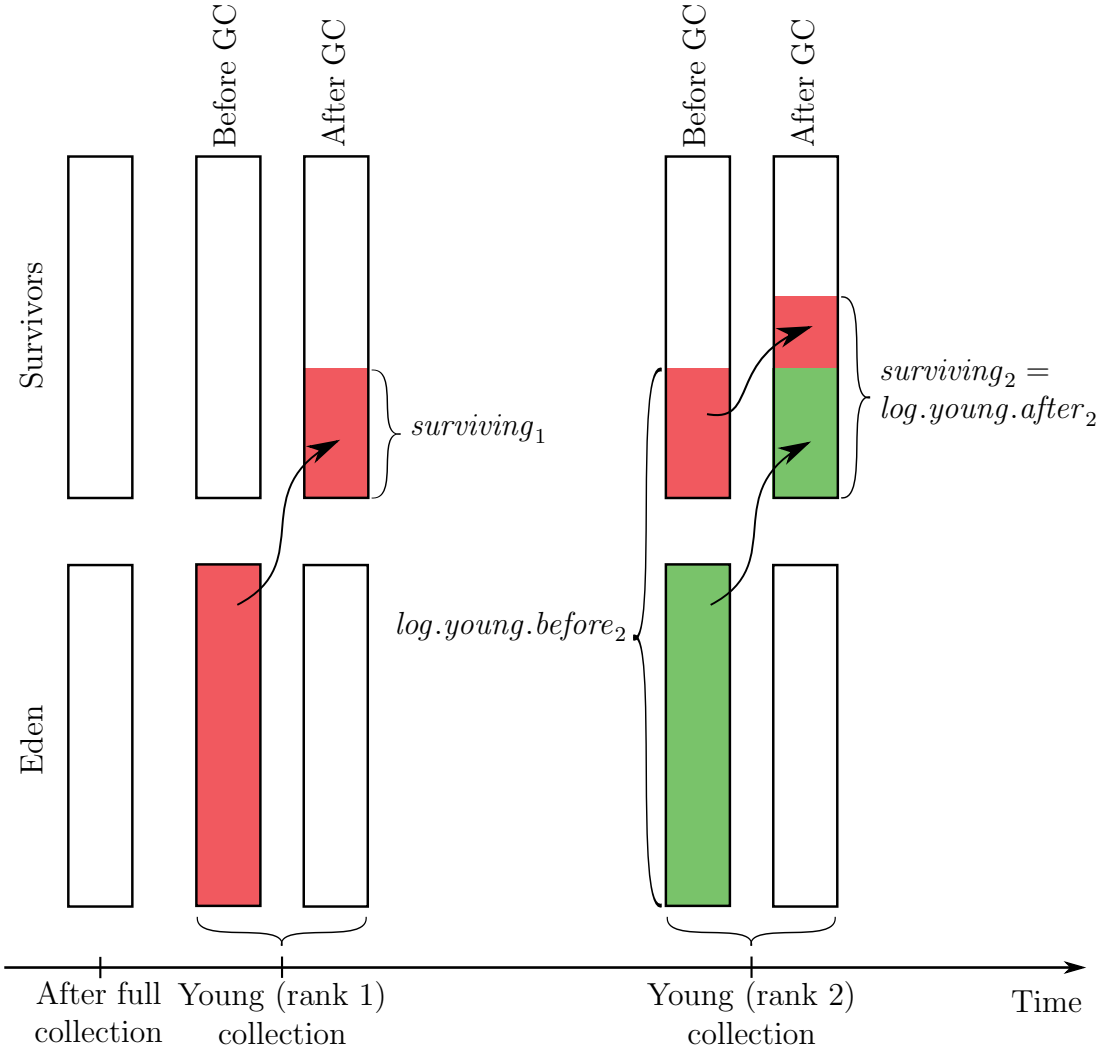


Figure 5.2: $surviving_2$ illustration

estimates $surviving_i$ for each $i \in 1 \ldots tenuring.threshold + 1$. We note that a premature promotion due to survivor overflow (§3) may introduce inflation in the estimate of the amount of surviving objects, we therefore omit such estimates and average over the remaining ones:

$$valid.surviving_i = \{surviving_i : log.young.after_i < max.size.survivor\}$$
$$surviving.average_i = \text{average}(valid.surviving_i)$$

5.3.2 Considering Additional Allocations

We now consider modifications that add allocations of short-lived objects (§8). In the text, we refer to the application without modifications as the *original* application (and original allocations, original collections and so on for artifacts present in the original application), and the application with envisioned modifications as the *modified* application (and modified allocations, modified collections and so on for artifacts not present in the original application).

To describe where the modified allocations happen, we execute the original (§10) application with minimalistic instrumentation that records information on free memory (§7) in all locations where the modified allocations are to be added.

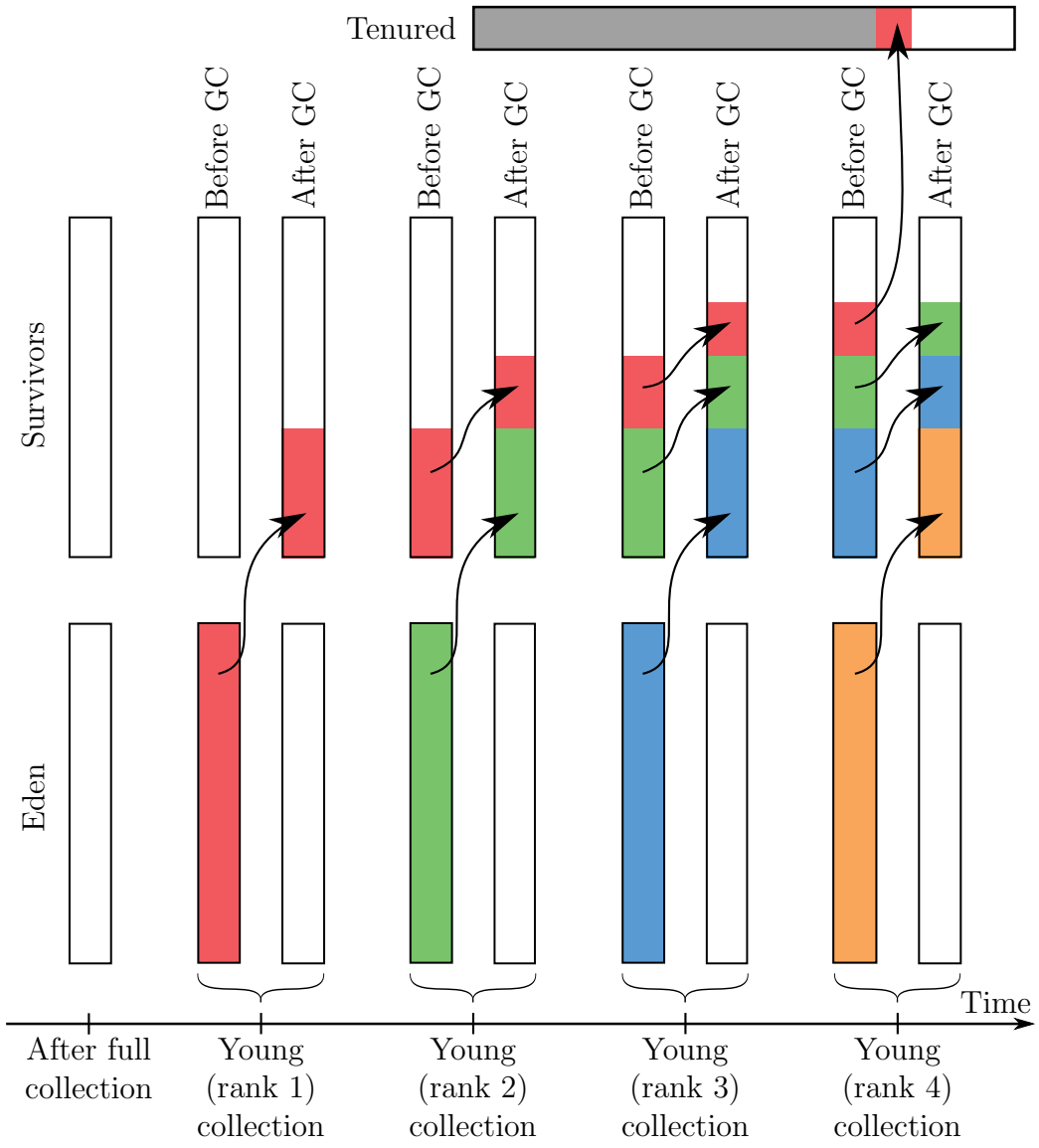


Figure 5.3: $surviving_i$ illustration

Merged with the record of the original GC behavior (§6), this forms the record of the original allocation behavior as the input of our model, which then estimates the modified GC behavior.

Triggering Young Collections. We pass sequentially through the record of the original allocation behavior, keeping track of the aggregate size of original and modified objects in eden, *in.eden.original* and *in.eden.modified*, and the aggregate size of objects in survivors, *in.survivor.original*. We only need to consider the original objects in survivors, because the modified objects are short-lived and therefore unlikely to survive (§8). We denote $in.young.original = in.eden.original + in.survivor.original$.

The modified allocations will cause the eden to fill up faster until a modified young collection is triggered, this is simply the moment when $in.eden.original + in.eden.modified$ reaches *max.size.eden*.

Estimating Surviving Amount. When the modified young collection is triggered, the young generation will contain both original and modified objects. To estimate the modified amount of surviving objects, we rely on the knowledge of the average amount of surviving objects in original collections of rank r, or $\mathit{surviving.average}_r$.

We define $\mathit{surviving.interpolated}(x)$ as an interpolation of $\mathit{surviving.average}$ for allocated amount $x \in 0 \ldots \mathit{max.eden.size} \cdot (\mathit{tenuring.threshold} + 1)$:

$$\begin{aligned} \mathit{surviving.average}_0 &= 0 \\ r.\mathit{complete} &= x \operatorname{div} \mathit{max.size.eden} \\ r.\mathit{partial} &= x \bmod \mathit{max.size.eden} \end{aligned}$$

$$\begin{aligned} \mathit{surviving.interpolated}(x) = \mathit{surviving.average}_{r.\mathit{complete}} + \\ + \frac{r.\mathit{partial}}{\mathit{max.size.eden}} \cdot \left(\mathit{surviving.average}_{r.\mathit{complete}+1} - \mathit{surviving.average}_{r.\mathit{complete}}\right) \end{aligned}$$

We illustrate the $\mathit{surviving.interpolated}(x)$ computation in Figure 5.4. On the horizontal axis we have allocated bytes, on the vertical axis bytes surviving after allocating x bytes since the last full collection. We know the $\mathit{surviving.average}_i$ values for the moments of young collections which happen after allocating $i \times \mathit{max.eden.size}$. These are tops of the bars in survivor spaces in the illustration, with the last one increased by the amount of data promoted to the tenured space (which can yield larger number than is the survivor size). We just connect these points and $(0, 0)$ point to get the desired $\mathit{surviving.interpolated}(x)$ function.

Although we define the $\mathit{surviving.interpolated}(x)$ function using the data from collections following the full collection we can apply it also for other collections. This comes from the fact that if we look at some later collection we move the 0 on horizontal axis forward to the moment of the end of collection $i - (\mathit{tenuring.threshold} + 1)$ and we know that all previously allocated objects are already promoted or dead.

Given an entirely stable allocation behavior (§9), we could set

$$\mathit{surviving.modified} = \mathit{surviving.interpolated}(\mathit{in.young.original})$$

using the interpolation directly. To support some fluctuations in survival behavior, however, we further adjust the estimate by looking at the original survival behavior in the nearest young collection. Specifically, for a modified young collection of rank r, we find the nearest original young collection of the same rank r. We then look at how the amount of surviving objects in this original collection differs from the average amount of surviving objects and adjust the estimate accordingly:

$$\begin{aligned} \mathit{surviving.original} &= \mathit{log.heap.after}_r - \mathit{log.heap.before}_r + \mathit{log.young.before}_r \\ \mathit{scale} &= \frac{\mathit{surviving.original}}{\mathit{surviving.average}_r} \\ \mathit{surviving.modified} &= \mathit{surviving.interpolated}(\mathit{in.young.original}) \cdot \mathit{scale} \end{aligned}$$

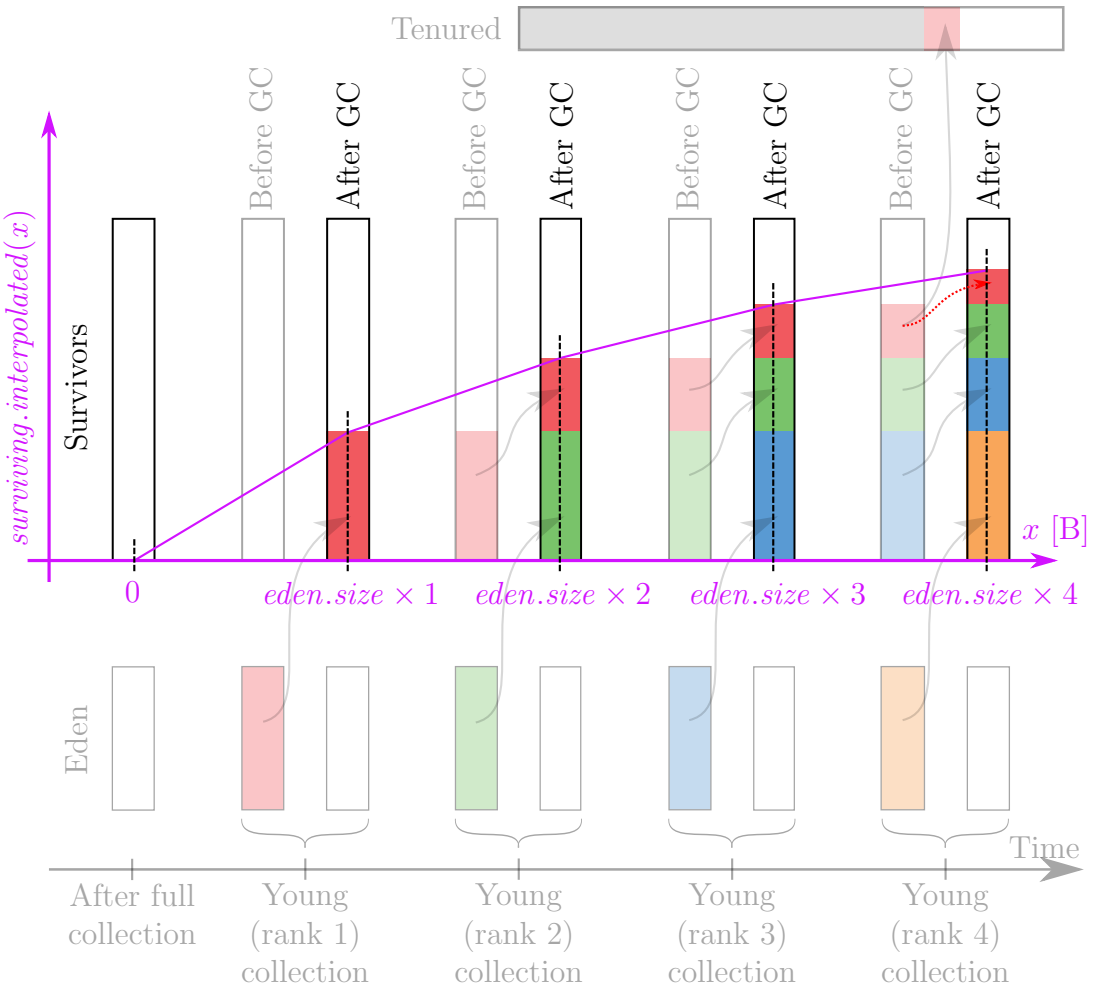


Figure 5.4: *surviving.interpolated*(x) (in magenta) illustration

The situation is depicted in Figure 5.5. We want to achieve the ratio between value from original collection and the value on the *surviving.interpolated()* curve (depicted green in illustration) is equal to the corresponding ratio in the modeled modified collection (blue brackets). This way, the resulting $surviving.modified$ amount is in the same relative distance from the *surviving.interpolated*() function as is the *surviving.original* amount.

Estimating Promoted Amount. After estimating the modified amount of surviving objects, we estimate the modified amount of promoted objects. Premature promotions aside, a modified young collection can only promote objects if its index exceeds the tenuring threshold. For such collections, we compute the promotion rate of the nearest original young collection:

$$promotion.rate = \frac{log.tenured.after_r - log.tenured.before_r}{surviving.original}$$

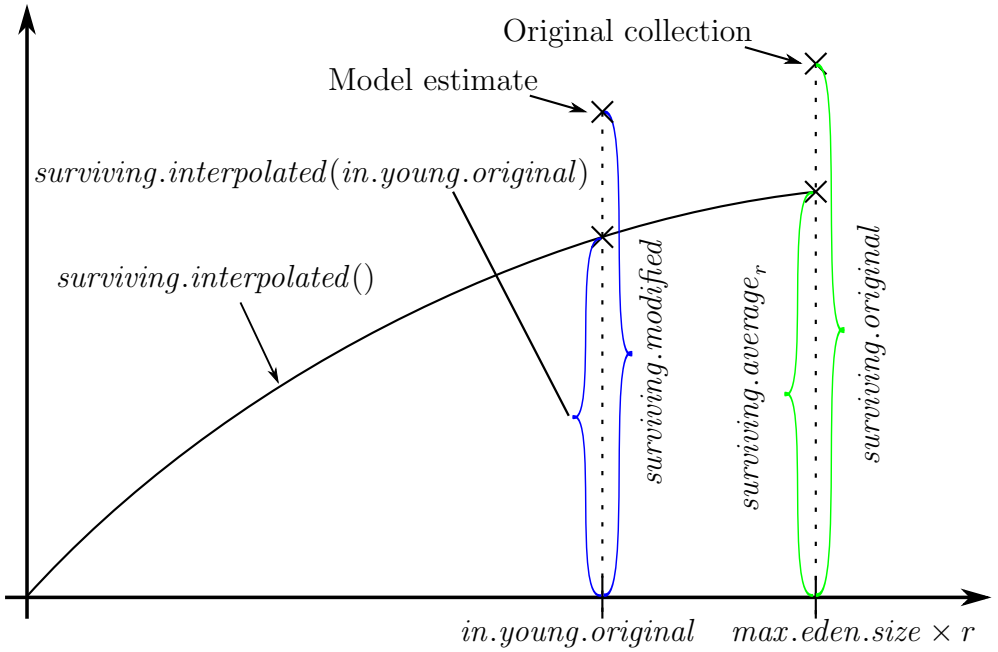


Figure 5.5: Surviving size calculation illustration

When the nearest original young collection involved premature promotions, we instead use the average promotion rate computed from *surviving.interpolated*:

$$surviving.all.lifetimes = surviving.interpolated(in.young.original_r)$$
$$surviving.except.oldest = surviving.interpolated(in.young.original_{r-1})$$
$$promotion.rate = \frac{surviving.all.lifetimes - surviving.except.oldest}{surviving.all.lifetimes}$$

The situation with premature promotion in original collection is illustrated in Figure 5.6. This would be the case with *surviving.original* exactly on the *surviving.interpolated*() function (with a different scale the situation is the same if the function is scaled). The intuition behind the calculation can be explained with help of the Figure 5.4, the calculation of the *surviving.interpolated* function. The promoted amount is depicted in red in the rightmost column. Given the stable allocation behavior (and therefore stable sizes of the bars of different age), it is clear that the promoted amount equals the difference between the *surviving.interpolated*() values for rank 4 and rank 3 collections in the illustration. For modified collection the situation is similar, only the bars are at different positions and therefore we use different parameters for *surviving.interpolated*() function.

From the difference we just calculate the promotion ratio and apply it to the already computed *surviving.modified*. This also causes the promoted size to be adjusted by the fluctuations in observed collection as the *surviving.modified* is scaled already.

Finally, we adjust the aggregate size of objects in survivors and in the tenured generation. The survivors will hold

$$\min(surviving.modified \cdot (1 - promotion.rate), max.size.survivor)$$

bytes, the rest is promoted.

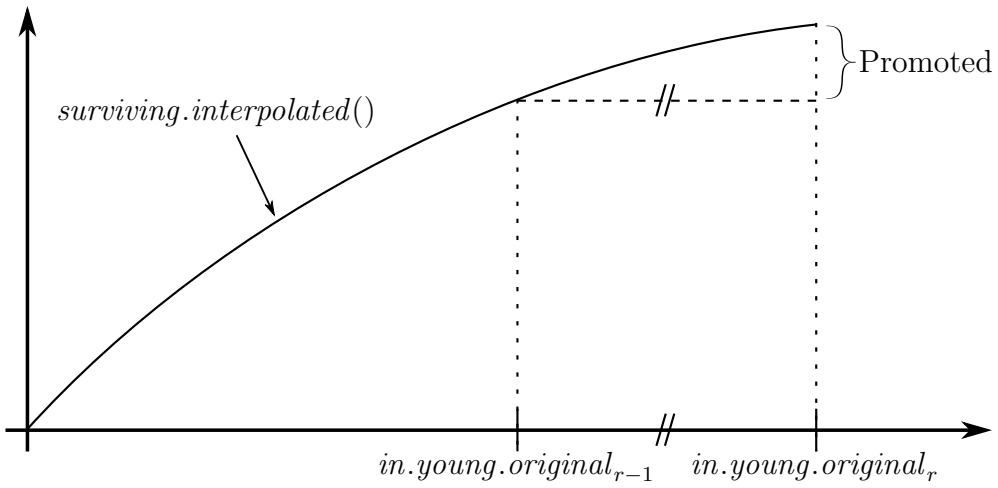


Figure 5.6: Promoted size calculation illustration

Triggering Full Collections. After estimating the promoted amount for the modified young collection, we update the weighted average of the promoted amount, which serves as the reserve for triggering full collection (§5). For this, we simply reproduce the formulas used in the virtual machine sources.

Finally, we test whether a modified full collection is triggered. If it is, we find the nearest original full collection and use the size of the tenured generation after this collection as the size of the tenured generation after the modified full collection. This estimate is possible because the modified objects are short-lived and the tenured generation therefore contains mostly original objects (§8).

5.3.3 Estimating Collection Time

Given the estimate of the modified GC behavior, we complete the model with estimates of the modified GC time. This is trivial for the full collections—although the modified collections may differ from the original collections in frequency, they traverse and collect mostly original objects in similar amounts. As a consequence, we estimate that each modified full collection takes about as much time as the nearest original full collection.

For the young collections, we rely on the empirical observation that the young collection time is strongly correlated with the number of live objects. We use total amount in place of total number of objects and estimate the modified time based on the time of the nearest original young collection with the same rank:

$$surviving.original_r = log.heap.after_r - log.heap.before_r + log.young.before_r$$

$$time.modified_r = time.original_r \cdot \frac{surviving.modified_r}{surviving.original_r}$$

5.4 Evaluation and Discussion

The primary goal of our evaluation is to understand and explain what makes or breaks the model. Towards this, we present and discuss the results of using the model on two workloads. With one workload, the model works reasonably well, especially given the quality of the inputs. With the other workload, the results

are notably worse. This is interesting because the workloads represent similar applications, but differ in how they satisfy the model assumptions on application behavior. In particular, the first workload has a stable allocation behavior, while the second workload has two alternating phases, each allocating objects with significantly different demographics.

5.4.1 Methodology and Metrics

Our evaluation is based on comparing the measured and predicted values of metrics that capture the amount of GC work. The key metrics are the total number of young and full collections and the total time spent doing young and full collections (in seconds). We also collect internal model metrics which serve as the basis for the high-level metrics—the average amount of data surviving young collections, the average amount of data promoted in young collections, and the average tenured generation occupancy before and after full collections (all in bytes). These enable better understanding of the results, especially in the cases where the model loses accuracy.

In each experiment, we first execute the original workload with a workload-specific GC configuration[5] that conforms to (§4). The JVM is instructed to produce a GC log (§6). The planned modification locations are instrumented per (§10), the instrumentation is carefully designed to avoid allocating any heap memory. The results of this run provide inputs for the model as well as baseline data for evaluating the real effect of the added allocations.

In the second step, the workload is modified to allocate more data at the designated locations and run using the same JVM configuration. The allocated data is a single integer array of configurable length, and is only used in the scope of the modified code, thus increasing the allocation rate of the workload without increasing the steady state live size. The results from this run provide data for establishing the ground truth regarding the effect of the added allocations.

Third, we solve the model using data from the original workload execution.

5.4.2 Workloads

Ideally, our evaluation workloads would be standard benchmarks. Unfortunately, this runs into difficulties—SPECjvm2008 does not exhibit an interesting allocation behavior in our context, SPECjEnterprise2010 is extremely unwieldy and not well supported with open source technologies (proprietary platforms restrict result publication), and the DaCapo [17] suite breaks our requirements with often rather low heap occupancy and distinct sawtooth patterns in the live size profile (owing to naive use of iterations) [49].

The `h2` workload from DaCapo comes closest to the live size stability assumption (§9), but is still rather uninteresting in a single iteration—with a reasonable[6] heap size, the workload does not trigger any full collection. We therefore use the `h2` workload with 400 iterations, default input size, 1 thread, and no forced GC. We also create a modified benchmark that performs additional allocations in

[5] We manually fix the min and max heap sizes, the size of the young generation space, the ratio of the eden and survivor spaces, and the tenuring threshold.

[6] Heap size 10 % above the minimal heap size required by the workload.

each transaction, with a total of 13 479 600 added allocations. The JVM configuration uses 256 MB tenured generation, 96 MB young generation (32 MB eden, two 32 MB survivor spaces), and a tenuring threshold of 7.

Even in the above configuration, the h2 workload still violates some of the model assumptions—the most notable being the presence of two distinct phases in every iteration. In the work phase, h2 allocates objects representing database records that contribute to the global state. In the cleanup phase, which restores the initial state of the database, h2 allocates only very short-lived objects that do not survive even one collection. Both phases are clearly visible in the GC log, and in the live size trace shown in Figure 5.7. Moreover, the workload design (together with multiple iterations) causes the full collections to synchronize with the end of some iterations (every second one in our case). The full collections always happen approximately in the middle of the cleanup phase, and changing the size of the tenured generation only influences their frequency, but not the point at which they occur during the iteration.

To evaluate the model on a workload that better conforms to the underlying assumptions, we created a benchmark simulating a simple university information system accessing an in-memory database. The benchmark, called dbart, operates on entities such as students, courses, schedules, and grades. Its live size trace is more stable, as shown in Figure 5.7. The modified benchmark performs additional allocations in the operation that records courses for a student, with a total of 5 935 084 added allocations. The JVM configuration uses 448 MB tenured generation, 192 MB young generation (64 MB eden, two 64 MB survivor spaces), and a tenuring threshold of 4.

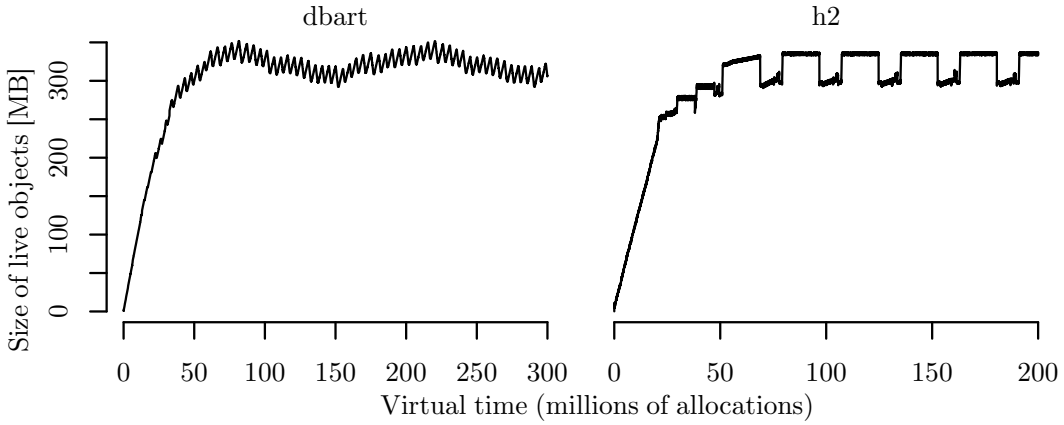


Figure 5.7: Live size, dbart and DaCapo h2 workload, partial

5.4.3 Measurement Platform and Results

We conducted the experiments on two different hardware platforms. The h2 workload was executed on a Dell PowerEdge M620 system[7] and OpenJDK 1.8.0_25 JVM[8]. The dbart workload was executed on a Dell PowerEdge 1955 system[9] and

[7] 48 GB RAM, two Intel Xeon E5-2660 (Sandy Bridge) chips, 16 processors, NUMA.
[8] OpenJDK Runtime (1.8.0_25-b18) and OpenJDK 64-Bit VM (25.25-b02).
[9] 24 GB RAM, two Intel Xeon E5345 (Clovertown) chips, 8 processors.

Oracle HotSpot 1.8.0_11 JVM[10]. In both cases, the JVM was configured to use a single GC thread to make the GC times stable.

To evaluate the model for different amounts of additional allocations, the modified workloads were executed in two configurations, using arrays of 2^{11} (2K) and 2^{12} (4K) elements as a base allocation unit. This corresponds to allocating 8 208 and 16 400 extra bytes (including object header and alignment), respectively, at each instrumented workload location.

The measurement and model evaluation results are summarized in Table 5.3. For each workload, the table shows the key metrics corresponding to the execution of the original workload, followed by results for the two modified workloads. For each modified workload, the table shows metrics obtained by measurement and by evaluating the model. Table 5.4 summarizes the accuracy of the model in form of prediction errors for the metrics that serve as a basis for calculating collection counts and durations. For each modified workload configuration, the table shows two prediction errors. The *error.wrt.base* is calculated as $\left|1 - \frac{modified.model}{modified.measurement}\right|$, and expresses the relative difference between the measured and predicted metrics for the modified workload. The *error.wrt.change* is calculated as $\left|1 - \frac{original.measurement - modified.model}{original.measurement - modified.measurement}\right|$, and expresses the error made in predicting the change in GC work. This error is not calculated for the tenured amounts before and after full collection, because the high-level metrics such as collection counts and times are influenced by the difference between the amounts before and after full GC, but not by the difference between the original and modified workloads.

5.4.4 Results Discussion

The results for the dbart workload are encouraging. The predicted young collection counts are very close to the measured values, which is expected [49]. The prediction accuracy for the survived and promoted amounts, summarized as averages in Tables 5.3 and 5.4, can be considered reasonable, as illustrated in Figures 5.8 and 5.9, which plot the individual predicted values.

The lower prediction accuracy for the young collection times and full collection counts can be primarily attributed to errors in predicting the survived and promoted amounts—these errors are additive and accumulate over all young collections, of which there are thousands. The errors also influence each other, e.g., over-estimating the promoted amount causes under-estimating the survived amount, which in turn results in under-estimating the total young collection time (which has a linear dependency on live size).

The accuracy of full collection count estimates is also influenced by the estimates of free space in the tenured generation, which is based on the estimated amounts of tenured objects before and after a full collection. We again consider these estimates reasonably accurate, as illustrated in Figure 5.10, complemented with Figure 5.11 where we plot how much space is available in tenured space for promotions between collections.

To explain why we consider the high-level results generally encouraging, consider the 4.82 % error in the prediction of the promoted amount for the 4K variant

[10]Java(TM) SE Runtime (1.8.0_11-b12), and Java HotSpot^{TM}64-Bit VM (25.11-b03).

of the dbart workload. To predict the full collection count exactly, the promoted amount would need to be predicted with error no more than 0.47 %, which we consider impossible given the model inputs.

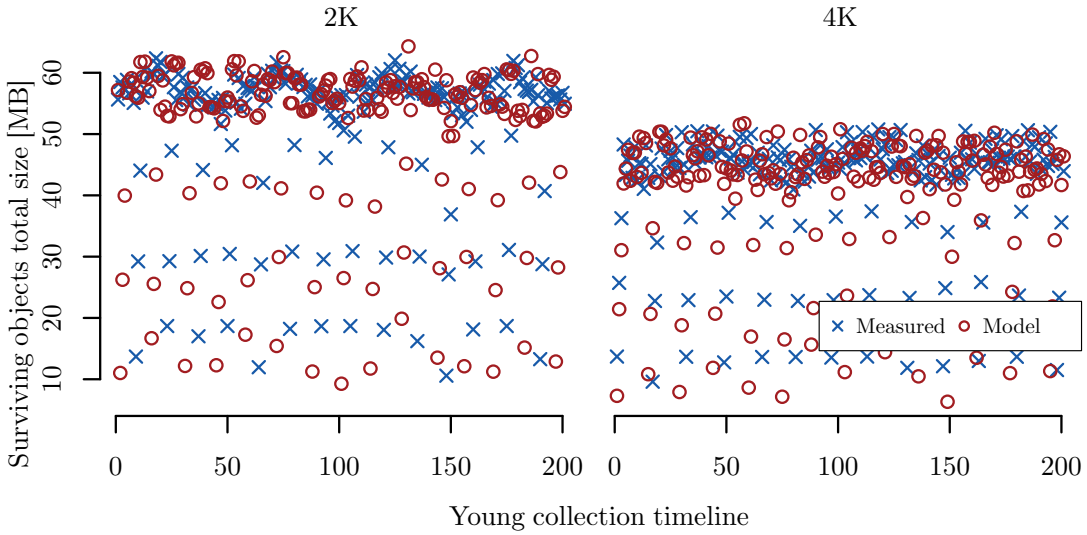


Figure 5.8: Size of objects in survivor space after young collections, dbart workload, partial

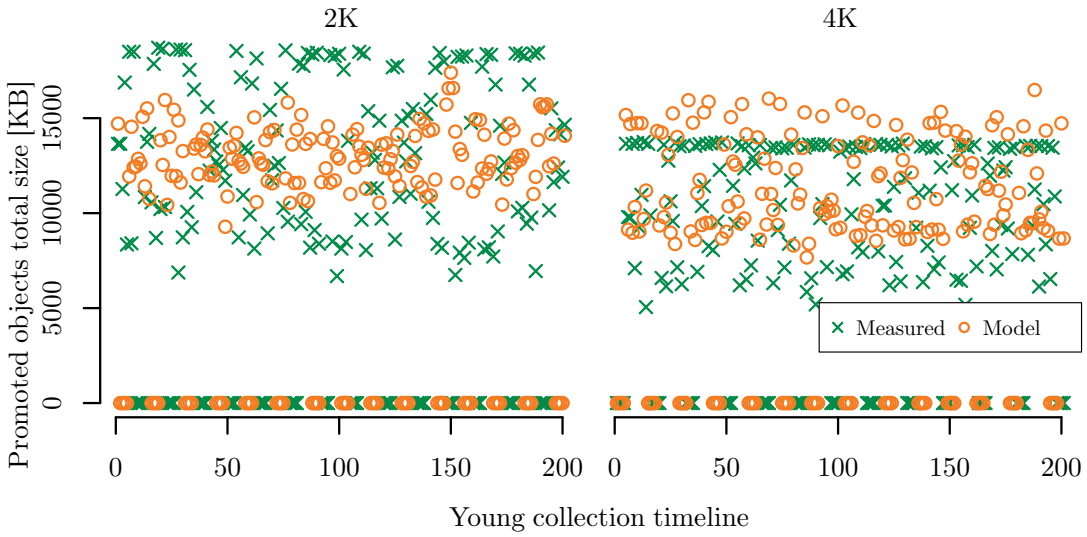


Figure 5.9: Size of promoted objects in young collections, dbart workload, partial

The results for the h2 workload are considerably less accurate. Similarly to the dbart workload, the individual predicted values of selected metrics are shown in Figures 5.12–5.15. The predicted full collection counts and times basically match the measured values, which we consider a coincidence of two errors canceling each other (over-estimating the promoted amount by 7 % and compensating by over-estimating the free space in the tenured generation).

The match in the tenured amount after full collection is not really surprising—the model estimates the value using data from the nearest full collection occurring in the original workload, and due to the full collections always occurring at the

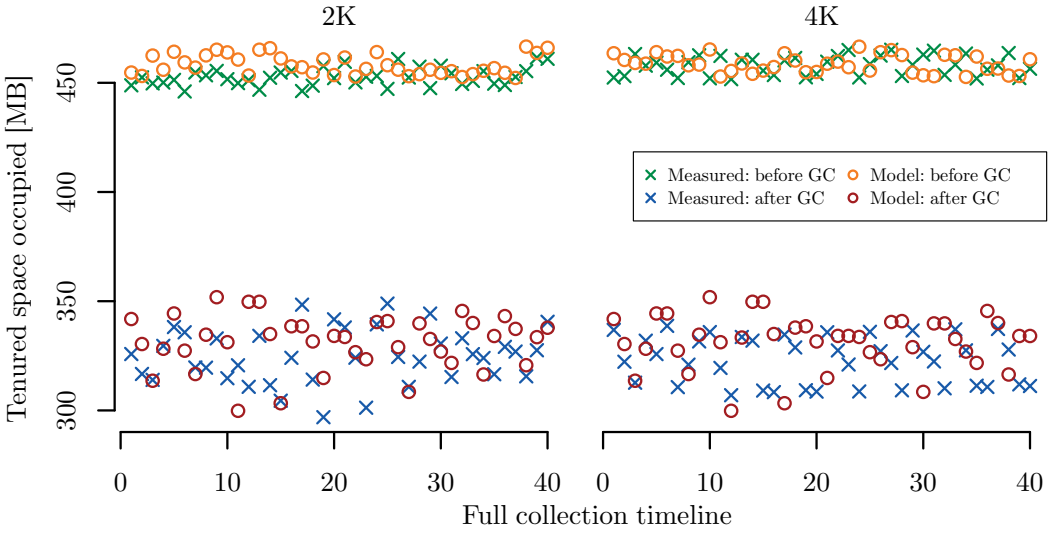


Figure 5.10: Tenured space sizes before and after full collections, dbart workload, partial

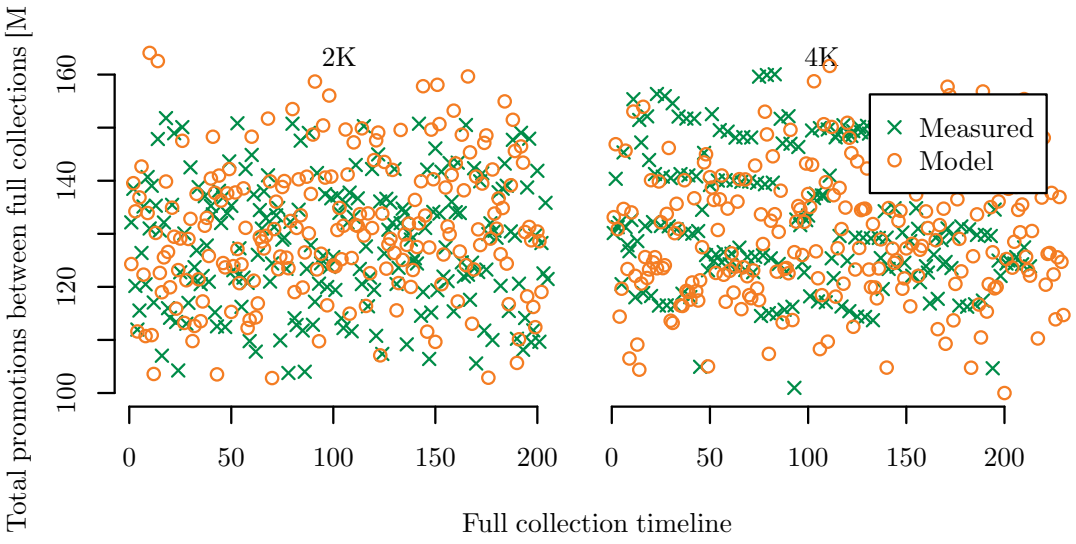


Figure 5.11: Space for promoted objects between full collections, dbart workload

same point in the cleanup phase, there is basically no room for observing different values. The tenured amount before full collection is predicted with reasonably low error, given the complexity of the tenured space reserve calculation.

The estimate of the young collection time is rather inaccurate. This is due to significant under-estimation of the survived amount, which is in turn caused by the presence of the alternating workload phases and the fact that a full collection always occurs at the same point in the cleanup phase. Because the cleanup phase allocates extremely short-lived data, the model observes $surviving_i \approx 0, i \in \{1 \dots 5\}$, which severely disrupts the *surviving.average* calculation—a major contributing factor to the survived amount estimation.

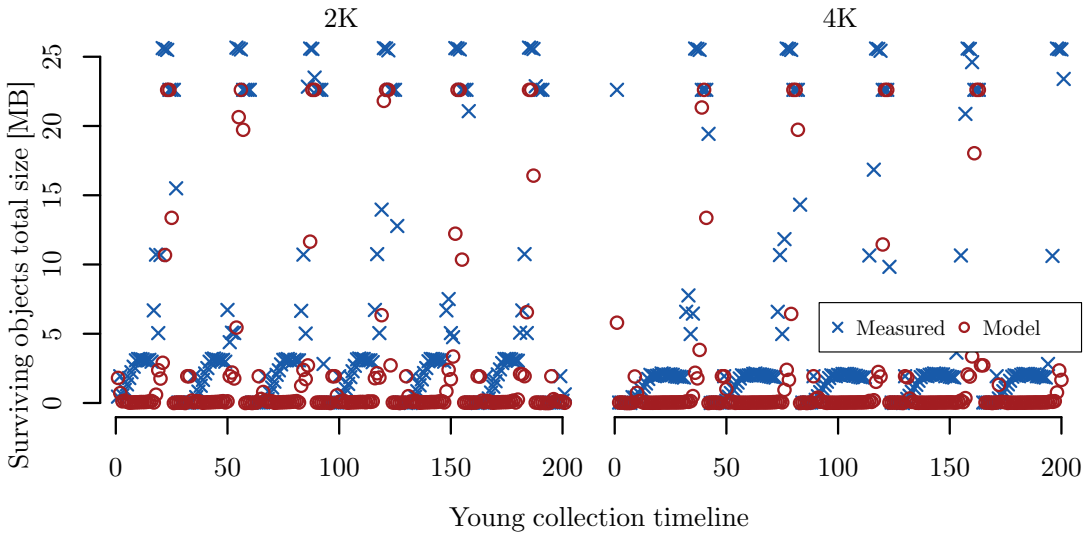


Figure 5.12: Size of objects in survivor space after young collections, h2 workload, partial

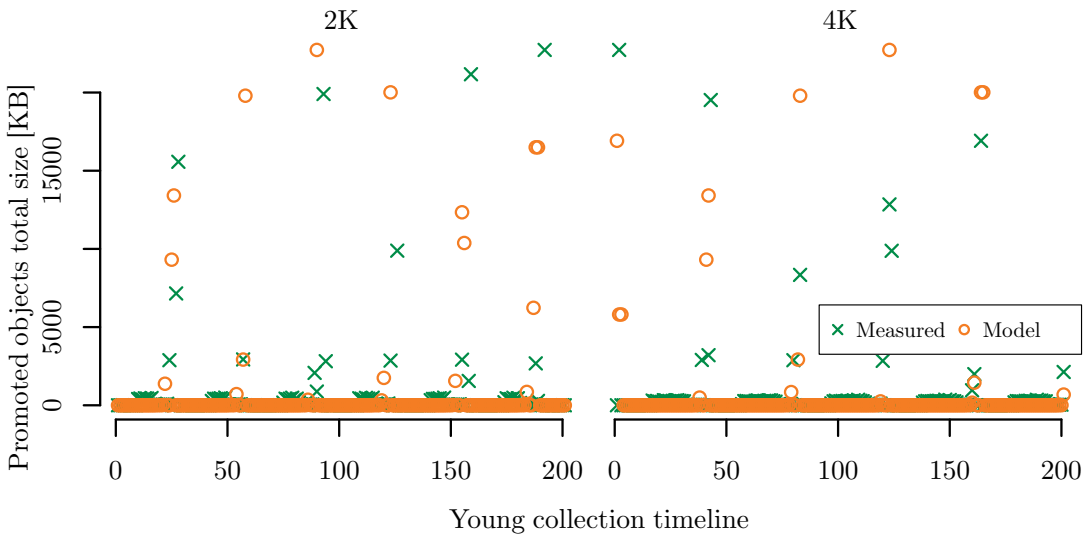


Figure 5.13: Size of promoted objects in young collections, h2 workload, partial

The *surviving.average* is used to compute *surviving.interpolated()* function, which in turn is used to estimate the amount of data surviving young collection.

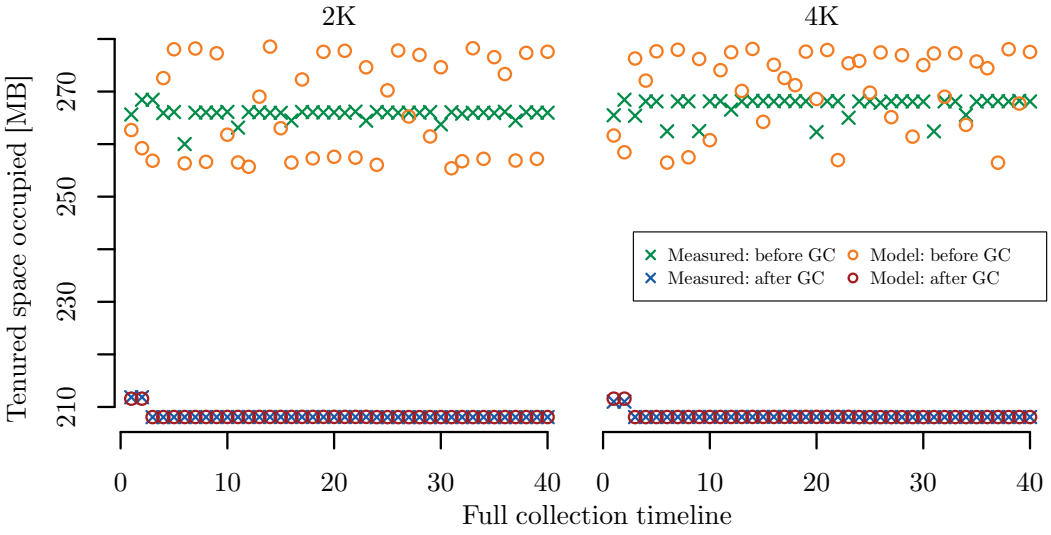


Figure 5.14: Tenured space sizes before and after full collections, h2 workload, partial

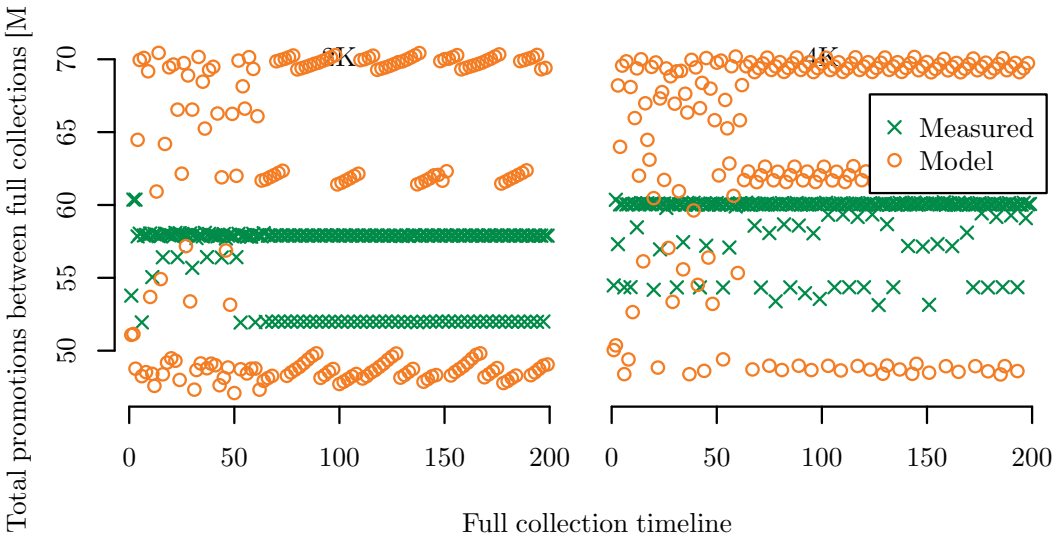


Figure 5.15: Space for promoted objects between full collections, h2 workload

As depicted in Figure 5.4, the function is constructed from several connected segments, each segment is linear. The meaning of the segment i is that the difference between its right and left end is the amount surviving i young collections. If the slope of the segment i and $i+1$ is the same, it means that each object surviving i collections will also survive $i+1$ collections—therefore if slope of segment i is bigger than slope of segment j, where $i > j$, some objects that did not survive j collections have to survive i collections, which is impossible. However, as we can see in Figure 5.16 where we display our *surviving.interpolated()* estimation for h2 workload, we observe exactly this anomalous situation. As opposed to h2, for the dbart workload the function is sound, as displayed in Figure 5.17.

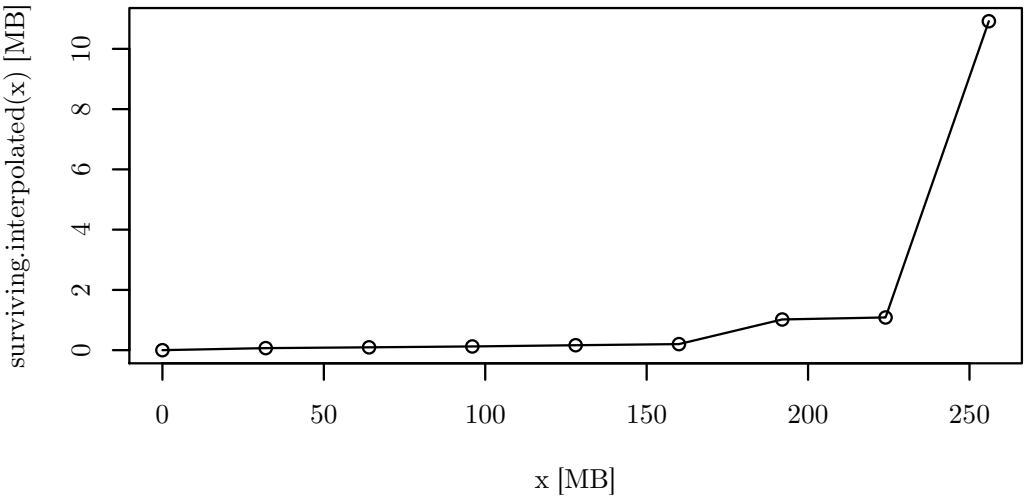


Figure 5.16: *surviving.interpolated()*, h2 workload

5.4.5 Linearity of Young Collection Times

To estimate times for young collections, we use linear dependency on size of live objects in the collection, with base being the time and live size in nearest original collection, as discussed in Section 5.3.3. Here we discuss validity of such approach.

For that, we measured the dependency of young collection time on live size in the young generation in our workloads. We show the results from dbart workload here, they are displayed in Figure 5.18. The linearity is very strong in the left part of the plot, while for the largest sizes the collections are slightly slower than the linear approximation from previous part of the plot would suggest. We believe this is caused by the fact that these slower data points come from collections with promotions, while the clusters on the left are from collections without promotions, because they represent collections with smaller rank.

It is clear the linear dependency is not perfect but we believe it is the best we can do with the model inputs we have. In reality the garbage collector performs depth-first-search traversal over the live objects in the young generation (plus objects that might be dead but have a reference from tenured space). This has

	Configuration	Young collection Time [s]	Young collection Count	Full collection Time [s]	Full collection Count	Survived young[1] [MB]	Promoted[2] [MB]	Tenured at full GC start[3] [MB]	Tenured after full GC end[4] [MB]
dbart	Original	1418.8	2128	1556.2	203	55.48	11.18	428.73	313.13
	2k – measurement	1629.7	2856	1563.1	206	48.64	8.92	431.88	309.81
	2k – model	1604.9	2854	1557.8	203	47.96	8.99	438.20	313.38
	4k – measurement	1744.7	3587	1617.0	214	40.15	7.74	436.52	308.67
	4k – model	1641.6	3579	1770.9	231	38.65	8.11	437.38	313.19
h2	Original	189.6	9762	196.4	200	8.54	1.05	249.08	198.54
	2k – measurement	234.0	13 395	197.4	200	6.59	0.81	252.30	198.53
	2k – model	70.2	13 057	196.4	200	2.42	0.87	254.73	198.54
	4k – measurement	255.2	16 405	197.2	200	5.47	0.70	254.94	198.53
	4k – model	68.2	16 348	196.4	200	1.84	0.75	258.98	198.54

[1] Total size of objects that survived young collection and stayed in survivor space (averaged over all young GCs).

[2] Total size of objects that were promoted to tenured space in the young collection (averaged over all young GCs).

[3] Total size of objects in tenured space at the moment when full collection started (averaged over all full GCs).

[4] Total size of objects in tenured space right after the full collection (averaged over all full GCs).

Table 5.3: Measured and modeled results

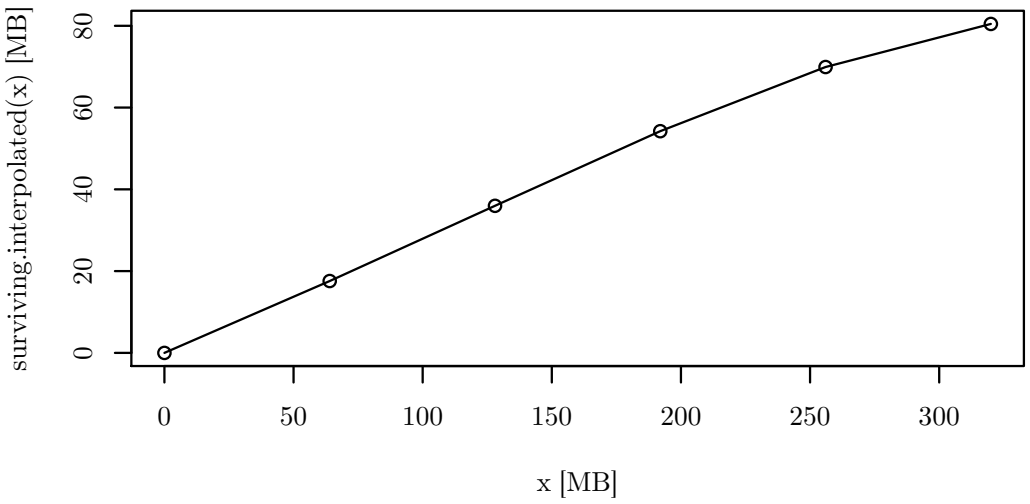


Figure 5.17: *surviving.interpolated()*, dbart workload

		Error wrt	Survived young GC [%]	Promoted [%]	Tenured at full GC start [%]	Tenured after full GC end [%]
dbart	2K	base	1.41	0.85	1.46	1.15
		change	10.04	3.34	—	—
	4K	base	3.75	4.82	0.20	1.47
		change	9.81	10.84	—	—
h2	2K	base	63.22	7.36	0.97	0.01
		change	213.86	25.00	—	—
	4K	base	66.40	7.45	1.59	0.00
		change	118.52	14.60	—	—

Table 5.4: Accuracy of the internal model metrics

linear complexity in number of traversed references (edges) plus visited objects (vertices). The work for each reference (bounded to references in young generation) is checking if the objects it is referring to were already marked and the work for each visited object is to set the mark and to copy the object either to survivor or to the tenured space. Moreover, it has to update references pointing to surviving young generation objects, which all changed their addresses. Affected are all references checked in the traversal and root references including those cross-generational. In our model we simplify this behavior to linear dependency on number of objects. We assume that the reference count per object is stable (§9) and therefore also depends on the number of objects within one application. We know such simplification can cause inaccuracies in model results, but more precise approaches would require knowledge of object references which is costly to obtain and would make the model computation impractically slow. It is also questionable if the improvements would be visible in the context of other model simplifications.

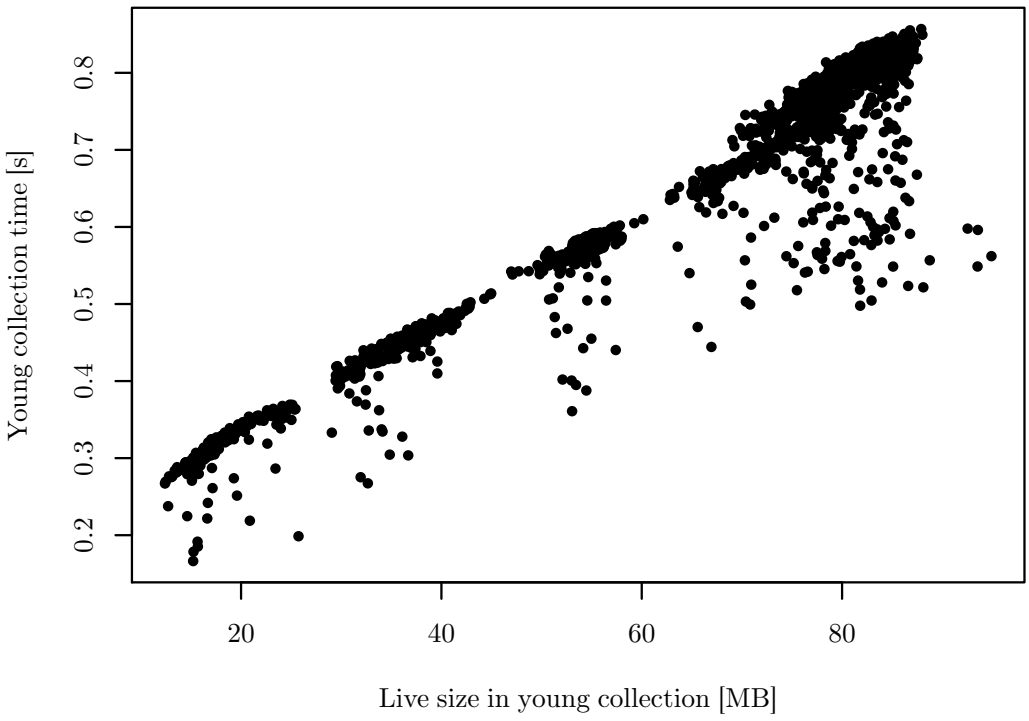


Figure 5.18: Dependency of young collection times on size of live objects, dbart workload

5.4.6 Considering Longer Lifetimes

We only considered code additions with short-lived objects in the model and evaluation. In this section, we describe what are the problems we have to face if we want to include also additions with longer lifetimes and sketch how the model and simulation could be modified to accommodate such objects.

Garbage collection performance and results depend largely on the lifetime of objects in the application. The model does not handle individual objects but we could perceive the single code addition as a set of smaller additions with every addition consisting of only objects that have the same lifetime. The other option is to handle individual objects in the simulator. In both cases, the behavior to model depends on object lifetime and it must be one of these cases:

1. Lifetime is such that the object is dead at the moment of the young collection. This situation does not differ from the model of pure short-lived additions therefore no extra handling is needed.

2. Lifetime is such that the object survives $1 \ldots$ *tenuring.threshold* young collections, meaning the object will not be promoted into the tenured generations. Here we do not consider cases with survivor overflows that would add another level of complexity to the problem. Such objects cause these effects:

- increase in survivor space occupation—object survives young collection and it is copied into one of the survivor spaces.
- increase in young collection time—collection algorithm has to traverse the object and copy it into the survivor space.
- it has no direct influence on tenured space and full collections, but indirect effect is possible in case of survivor overflows. The occupation of survivors has no impact on young collection frequency (it is only triggered when eden space is full) and if there is no overflow, only the objects that would end up in the tenured generation anyway are promoted. We skip the overflow case here because it is too complex, for example also the traversal order can be important.

3. Lifetime is big enough for object to reach tenured generation. Again, we ignore cases when the object can reach the tenured space because of overflow in the survivor space. Effects on young spaces and collections are the same as in case 2, but it also has effect on all full collections while the object is still alive:

   - increase in tenured space occupation—causing more frequent full collections because there is less space left for allocations.
   - increase in full collection time
   - possibly more inter-generation references—if the object from tenured space points to some objects in young generation the collector has to remember these references and include them in young collection roots. If an object in tenured space with such remembered references dies, the target of that reference and transitively also all objects that are referenced by that object are considered live in young collections (until the next full collection), even though they are already dead from the application perspective. This can cause bigger surviving sizes and results in longer young collection times and possibly an earlier full collection if survivor overflows.

5.4.6.1 Model Modifications Necessary

Now we discuss the model and input collection modifications necessary to include non-short-lived objects into the computation. First, we need to quantify lifetime of the objects, because most of the collector decisions are based on reachability which is closely related to lifetimes. Here we propose two formats, both suitable for different kind of additions and situations.

Format 1: The first option is lifetime trace with the time unit being the allocated bytes, which is common in GC related literature [36, 15]. Ideally, we would use the trace to identify moments in the original application where objects from the additions die. It is not realistic when considering an arbitrary addition, because for that we would need to measure the complete application and therefore the model could provide no benefit in design phase. On the other hand, if the objects allocated in addition are manipulated only in code of other additions, we could count the lifetime only within additions. With this scenario, we could create an application that consists from additions only, in the same order and counts as

in instrumented original application, and there we could measure lifetimes which could be then applied to the simulated model. This would rely on the assumption that the trace we collect from the application is stable and representative enough for possible executions.

Once we have the trace, the simulator can decide into which lifetime class the object belongs to and modify the collector behavior accordingly: the simulator stores the information in what space the object is, moves it between spaces as required and deletes the object when its lifetime has expired. One possible simplification is the option not to record exact lifetime for each object, but only the number of addition invocations it survives.

In the model we treat additional allocations atomically and as short-lived—meaning the objects are always dead at the moment of the collection. This can cause imprecision especially for additions that have large state. If the collection happens inside the addition's code the objects that are live increase the young collection time and survivor occupancy but the model ignores this. With lifetime trace of the addition available it would be simple to improve this behavior.

Format 2: The second lifetime option is direct partitioning of the addition's objects into lifetime classes. The envisioned partitions would be three: short-lived, long-lived and permanent, reflecting generational hypothesis and common usage of static or permanent objects. The model input could then be extended with the amount of short-lived and long-lived objects in each addition, along with average total size of long-lived data on the heap in the steady state. The amount of long-lived data should be relatively constant—for the application to be stable the allocation rate and death rate has to approximately equal, otherwise the application would end up without any objects or with full memory. For the same reasons permanent objects should be allocated only in initial phase.

The effects of short-lived partition can be modeled as before without modifications. Permanent partition, after initial phase, will have an effect of increasing size of the heap after full collection (causing higher frequency) and increase in full collection time. For the long-lived partition, if we know its size in each addition and how much live data from it resides on the heap, we can easily calculate how much addition invocations such objects survive (by simply dividing size on the heap by size from one invocation). Then the simulator can use this as lifetime estimate for those objects and handle the simulated collections accordingly.

Both lifetime format options leave the problem with inter-generation references unresolved. To tackle that the simulator needs to know about the references in the application which makes input collection and simulation much more complex, again. The simulator might incorporate one of the reduced input methods presented in Section 4.4 but the possible improvement is questionable and we decided to neglect the inter-generation references.

5.4.6.2 Implementation Concerns

In this section we have described possible approaches for extending the model to include also long-lived objects. However, we decided not to implement objects with longer lifetimes and now we explain why. The reasons lie with the input data collection and runtime needed to simulate the model.

The initial idea was to see how a model that takes input that is easy to collect and it is quick to calculate will behave. For longer lifetimes, we definitely need some lifetime measurement and that is nowhere near being fast and simple, as already mentioned in Section 4.3.2. Collecting lifetimes for applications running in order of minutes takes days and produces gigabytes of data. Moreover, with present infrastructure the data collected is not reliable because of the problem with stack allocations caused by the escape analysis—we are unable to observe real behavior on the heap because some objects that are allocated on the heap from the programming language perspective never reach the heap as seen by the virtual machine (details in Section 4.3.3).

For the lifetime format 2 (using lifetime partitions) it is unlikely to have an automated measurement approach working with just isolated additions. It would require expert judgment to separate the objects into the three partitions and estimate the average live size of long-lived objects. It would be easier if we had a different application with addition already included. There could work an approach that would observe when the objects from individual addition invocations are collected (using JVMTI object tags and deallocation events) to estimate which objects are long-lived or permanent and also how much live data from long-lived partition is on the heap. Unfortunately, this is not very lightweight approach, again, because it requires complete application and moreover the problem with stack allocations applies, too.

Then when we would obtain the input data, we would need to simulate the model. This will very likely take a long time, because the simulator needs to work with individual objects.

These challenges considered, we decided to stick with short-lived additions.

Our work is provided together with complete data and tools, available at http://d3s.mff.cuni.cz/resources/epew2015.

5.5 Summary of Chapter 5

The model in this chapter is in great contrast with the model presented in previous chapter. First, it is using only easily obtainable input—just GC log and trace of the additions extended with free heap space information. Second, the simulation is fast, typically running only a fraction of the time of the full application. This comes with the cost at expressiveness, however, the model can only capture constrained situation with code additions allocating short-lived objects only and requires the target program to have stable allocation behavior.

In the previous chapter, we have also seen how difficult and costly accurate GC modeling is. With our partial model in this chapter we have observed much better accuracy, especially in metrics that the model directly calculates. Unfortunately, most important high-level metrics still have their share of imprecision, mainly caused by their additive nature—the model would have to estimate its intermediate results with unrealistic accuracy.

The model is also sensitive to violations on stable allocation behavior assumption. We observed how the model accuracy suffers when the application executes in certain type of phases that synchronize the GC into specific points in execution. This distorts the information in the GC log in a way that is critical to our estimation on application objects' lifetime and survival behavior.

At the end, we think the model can find its usage in situation where implementing full new version of the software is too expensive and benchmarking is therefore difficult. Such situations include incorporating new features into software with limited GC budget and the developer needs to decide what implementation to use or into which code locations the feature will be implemented.

Chapter 6

Conclusion

We conclude the thesis by looking back at the goals presented in Section 1.4:

Goal 1: *Examine the nature of collector overhead.*

Goal 2: *Understanding the contribution of algorithm and implementation.*

Goal 3: *Insight with reasonable information.*

Goal 1: Based on the knowledge of the garbage collection principles and algorithms, we envisioned several factors that should influence GC overhead: allocation speed, maximum heap size, number and size of live objects, depth of the heap and ratio of short-lived to long-lived objects on the heap. We have designed three artificial workloads and ran an extensive series of experiments to determine if and how those factors influence collector overhead.

In most cases we observed the expected behavior: with faster allocations and larger heap we have seen higher collection overhead, with bigger maximum heap size the overhead was usually lower. Somewhat surprisingly, we did not see much influence on overhead when varying the size of objects or the depth of the heap.

However, the effect of the factors cannot be easily generalized because the differences in the shape of plots for different workloads or VM settings are too big. Seeing the data leads us to believe that the black-box approach will not work for garbage collection overhead. Of course, this could be caused by incorrect choice of factors but we consider that unlikely.

To consider if the treatment of garbage collection as a constant background factor is reasonably accurate, we can look into the individual plots presented in Chapter 3. In many cases we can see that the measured overhead for neighboring configurations is very similar (or increasing linearly in case of allocation speed plots) and that would justify simplifying the GC in performance models. However, in almost all plots we can see a rapid change in overhead at some point. Therefore we think that models should include the garbage collection at least at the level that would allow to detect such turning points.

The overhead of the collector reached 40 % even in workloads without excessive allocation speed and with plenty of headroom for the collector to operate in (i.e. in Figures 3.16 and 3.12). We consider such overhead to be too high to ignore in any model that has an ambition to predict performance with reasonable accuracy.

Goal 2: We defined a model of both a simple one-generation and a common two-generation collector. These models are based on knowledge of algorithms forming such collectors and also other basic internals. Therefore we think the model assumptions are more detailed than what is the average knowledge of GC principles we can expect from a developer who is not a GC expert. The performance predictions of such model then represent what the developer who has similar knowledge can anticipate when the circumstances are favorable.

We ran a series of experiments requiring huge amounts of input data (up to gigabyte per second of workload execution) and taking months to complete to evaluate the accuracy of the models. In general we observed good accuracy for one-generation models and very good accuracy for young collections in two-generation model. For full collections in two-generation collector the story is completely different, however. We've seen acceptable accuracy for some of the benchmarks but too often the accuracy is not good. We therefore conclude that knowing the algorithm is not enough to understand the collector performance—the specific details of the implementation and non-determinism have too much influence on the performance.

As an example of such detail we can list the inter-generation references. Generally, the platform will not expose the information in which generations the objects are allocated. However, writing a reference that crosses generation boundary can have different effects on GC performance than writing a reference within one generation. The cross-generation reference from older to younger generation has to be recorded in the remembered set (in case of HotSpot VM this brings no extra overhead—all reference updates are recorded in the card table) and, more importantly, if the source object dies while the reference is still there, it causes the subgraph pointed to by that reference to be considered live in young collections even if those objects are otherwise dead. In turn, those objects will make young collections slower and eventually get promoted into the old generation, causing the next full collection to happen sooner—all of this without any reasonable control of the developer.

In further analysis we found situations that are completely counterintuitive from the perspective of an application developer. For example making all objects smaller by 20 % can have no effect on the number of full collections. Without a doubt, it would be a lot of work to optimize an application so that it shrinks all the objects by such an amount and yet there may be no performance benefit in terms of full collections. In this thesis, we have also seen data showing that giving the collector more memory to operate in can actually cause higher collector overhead (Figure 3.14)), which is not only counterintuitive, but also contradicts previous work [15].

In the context of presented findings and our experience from experimentation, we have collected advice for the developers on what to be careful of when interested in predictable GC performance. First, this includes the escape analysis, which can cause some of the objects to be allocated on the stack (scalar replacement optimization) instead on the heap. Especially when working on dynamic analysis tools tracking object allocations, observation can effectively disable stack allocations making the collections more often. Also such seemingly harmless operations like debug logging can change the allocation behavior—passing an object that could be allocated on stack to a logging method will probably cause the

object to escape and therefore move the allocation to the heap.

Our second recommendation is useful especially for benchmark design. It is advisable to avoid scaling patterns with iterations at the end of which there is a large drop in the amount of live objects. This causes sawtooth-like changes of the live data volume in time and can cause synchronization of the full collections having highly counterintuitive effects, such as making the number of full collections close to certain fraction of the number of iterations. In the case of an application developer, if her application has sawtooth-like live data pattern, she may expect lack of sensitivity to some memory usage optimizations. We have also seen situations where she could increase object sizes without extra penalty on full collection frequency, which seems to be good for her first, but it also means the collector was not performing optimally at the beginning.

Our third recommendation is to not use the ergonomics when seeking reasonable GC behavior. We've seen that simple workloads like doubly linked list modifications can have large variation in observed GC performance due to ergonomics (Figure 3.10). Although the prospect of automatically good performance is tempting, it will also give suboptimal performance in many cases. The developer or administrator who cares for best performance should therefore tune the generation sizes manually.

Goal 3: As a result of the investigation related to previous goals, we have concluded solution for general prediction of collector performance is unlikely to exist. We therefore looked for situations in which we can provide useful performance advice. One such scenario is code modification where we want to insert short-lived object allocations into specific place(s) of an application operating in a steady state. We only require the developer to be able to identify the code locations where he intends to insert the code and estimation of the object sizes. With the help of a lightweight instrumentation and GC logs we can estimate the change of garbage collection performance with reasonable accuracy.

To support the prediction, we defined a collector model based on similar principles as full-simulation models from Section 4.3, but concentrating only on additions allowed us to estimate values critical for accurate predictions from observation of an unmodified application. Our model then tells us what is the effect of additions on these values. We especially appreciate how the amount of input data and simulation runtime is in orders of magnitudes smaller when compared to the full models from Chapter 4—tens of megabytes vs. tens of gigabytes and tens of seconds vs. hours.

Model accuracy can be hindered by certain allocation patterns of the original applications, like phases with significantly different allocation behavior or rapid changes of amount of live data (corollary of benchmarks problems mentioned with Goal 2). We also explained how the requirement on having only short-lived allocations is connected to practical advantages of the approach and how lifting these requirements would again make the data collection very difficult, time consuming and error-prone. It also increases the model input size significantly and uses more time for simulation while the accuracy can be lower.

We presented our research in response to this goal in Chapter 5.

Overall, we see that despite the progress made in this thesis, garbage collection performance remains a factor that is difficult to incorporate into software performance engineering. We can identify many situations that significantly influence GC performance and therefore deserve developer attention. We can also model collector behavior, however, we know that without an impractically large set of inputs or constraints on particular use case, we have to expect rather low accuracy. What we would still need is some guarantees of stability—where lower overhead estimation accuracy can be accepted and compensated e.g. by calibration, the fact that sometimes GC overhead displays surprising changes is uncomfortable. In this sense, collector overhead is similarly brittle as other chaotic dynamic systems.

Finally, we should also point out that the problem can be approached from a different side—rather than trying to understand and model the performance of a highly complex memory management infrastructure, we can attempt to modify the GC implementation to avoid some of the behavior aspects that are most difficult to model. Given that collector implementations are not brittle on purpose, this would likely imply trading some of the collector performance for stability. More, defining some metrics that would permit evaluating the trade-offs might be an interesting venue to pursue—after all, similar trade-offs can be found in many places of todays platforms, such as JIT, and making an informed decision about the split between performance and stability seems to be a better option than discovering various performance anomalies by trial and error.

Bibliography

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 224–236, New York, NY, USA, 2004. ACM.

[2] A.-R. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Washington, DC, USA, 2004. IEEE Computer Society.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), Jan. 2000.

[4] V. Babka, L. Bulej, M. Děcký, J. Kraft, P. Libič, L. Marek, C. Seceleanu, and P. Tůma. Resource usage modeling, Q-ImPrESS deliverable 3.3. http://www.q-impress.eu, February 2009.

[5] V. Babka, P. Libic, and P. Tuma. Timing penalties associated with cache sharing. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–4, Sept 2009.

[6] V. Babka, P. Libič, T. Martinec, and P. Tůma. On the accuracy of cache sharing models. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 21–32, New York, NY, USA, 2012. ACM.

[7] V. Babka, L. Marek, and P. Tuma. When misses differ: Investigating impact of cache misses on observed performance. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 112–119, Dec 2009.

[8] V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 77–96, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] V. Babka and P. Tůma. Can linear approximation improve performance prediction? In *Proceedings of the 8th European Conference on Computer Performance Engineering*, EPEW'11, pages 250–264, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] D. F. Bacon, P. Cheng, and S. Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 23–34, New York, NY, USA, 2012. ACM.

[11] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 207–235, London, UK, UK, 2001. Springer-Verlag.

[12] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 1–10, New York, NY, USA, 2010. ACM.

[13] F. Bause. Queueing Petri nets – a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23. IEEE, 1993.

[14] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, Jan. 2009.

[15] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1):25–36, June 2004.

[16] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[17] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[18] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[19] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[20] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970.

[21] C.-Y. Cher, A. L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 199–210, New York, NY, USA, 2004. ACM.

[22] T. W. Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[23] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, Dec. 1960.

[24] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 105–114, New York, NY, USA, 2000. ACM.

[25] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording microprocessor history. *Communications of the ACM*, 55(4):55–63, Apr. 2012.

[26] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[27] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, Sept. 1976.

[28] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 92–115, London, UK, UK, 1999. Springer-Verlag.

[29] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, Nov. 1978.

[30] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, Nov. 1969.

[31] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM '00, pages 111–120, New York, NY, USA, 2000. ACM.

[32] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[33] J. Happe, D. Westermann, K. Sachs, and L. Kapová. Statistical inference of software performance models for parametric performance completions. In *Proceedings of the 6th International Conference on Quality of Software Architectures: Research into Practice - Reality and Gaps*, QoSA'10, pages 20–35, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] B. Hayes. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 33–46, New York, NY, USA, 1991. ACM.

[35] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[36] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006.

[37] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM.

[38] J. Hruska. The death of CPU scaling: From one core to many – and why we're still stuck. http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck, January 2012.

[39] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.

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

[41] R. E. Jones and C. Ryder. A study of Java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

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

[43] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS) 2005*, pages 484–490. SCS, 2005.

[44] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM.

[45] K. Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models Using Static and Dynamic Analysis*. The Karlsruhe series on software design and quality. KIT Scientific Publ., 2012.

[46] M. Kuperberg, K. Krogmann, and R. Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 48–63, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, Jan. 2006.

[48] P. Libič, P. Tůma, and L. Bulej. Issues in performance modeling of applications with garbage collection. In *Proceedings of the 1st International Workshop on Quality of Service-oriented Software Systems*, QUASOSS '09, pages 3–10, New York, NY, USA, 2009. ACM.

[49] P. Libič, L. Bulej, V. Horký, and P. Tůma. On the limits of modeling generational garbage collector performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 15–26. ACM, 2014.

[50] P. Libič, L. Bulej, V. Horký, and P. Tůma. Estimating the impact of code additions on garbage collection overhead. In *Proceedings of the 12th European Conference on Computer Performance Engineering*, EPEW'15, Berlin, Heidelberg, 2015. Springer-Verlag.

[51] P. Libič and P. Tůma. Java garbage collector performance measurements. In J. Šafránková and J. Pavlů, editors, *WDS'09 Proceedings of Contributed Papers: Part I – Mathematics and Computer Sciences*, pages 34–40. MATFYZPRESS, June 2009.

[52] Y. Ling, T. Mullen, and X. Lin. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, 34(2):42–55, Apr. 2000.

[53] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, Dec. 1992.

[54] 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 *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM.

[55] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[56] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 245–260, New York, NY, USA, 2007. ACM.

[57] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[58] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[59] S. O'Grady. The RedMonk programming language rankings: June 2015. http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/, June 2015.

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

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

[62] Oracle. Java microbenchmarking harness (OpenJDK: jmh), 2014. http://openjdk.java.net/projects/code-tools/jmh/.

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

[64] Q-ImPrESS Consortium. Q-ImPrESS project. http://http://www.q-impress.eu.

[65] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable production of complete and precise GC traces. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 109–118, New York, NY, USA, 2013. ACM.

[66] R. Shahriyar, S. M. Blackburn, and D. Frampton. Down for the count? Getting reference counting back in the ring. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 73–84, New York, NY, USA, 2012. ACM.

[67] A. Shipilëv. Java Microbenchmark Harness (The Lesser of Two Evils). Presentation at Devoxx, http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf, 2013.

[68] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008/, 2008.

[69] Sun Microsystems, Inc. Garbage collector ergonomics. http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html, 2004.

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

[71] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[72] TIOBE Software. TIOBE index for august 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, August 2015.

[73] K. Ueno, A. Ohori, and T. Otomo. An efficient non-moving garbage collector for functional languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 196–208, New York, NY, USA, 2011. ACM.

[74] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[75] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 1–9, New York, NY, USA, 2009. ACM.

[76] J. Weizenbaum. Recovery of reentrant list structures in SLIP. *Commun. ACM*, 12(7):370–372, July 1969.

[77] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[78] 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), Sept. 2005.

[79] X. Zhang and M. Seltzer. HBench: JGC - an application-specific benchmark suite for evaluating JVM garbage collector performance. In *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*, COOTS'01, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

List of Figures

List of Tables