The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors

Philipp Lengauer
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Garbage collection, if not tuned properly, can considerably impact application performance. Unfortunately, configuring a garbage collector is a tedious task as only few guidelines exist and tuning is often done by trial and error. We present what is, to our knowledge, the first published work on automatically tuning Java garbage collectors in a black-box manner considering *all* available parameters. We propose the use of iterated local search methods to automatically compute application-specific garbage collector configurations. Our experiments show that automatic tuning can reduce garbage collection time by up to 77% for a specific application and a specific workload and by 35% on average across all benchmarks (compared to the default configuration). We evaluated our approach for 3 different garbage collectors on the DaCapo and SPECjbb benchmarks, as well as on a real-world industrial application.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory Management (Garbage Collection)*

General Terms

Performance, Experimentation, Measurement

Keywords

Garbage Collection, Configuration, Optimization, Java

1. INTRODUCTION

Garbage collection (GC) relieves programmers from reclaiming unused heap objects manually. This convenience has led to a wide-spread use of managed execution environments. Moreover, compacting garbage collectors allow for faster allocations because allocating an object is as simple as appending it to the end of the used heap, making expensive searches for a fitting memory block unnecessary.

ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2568091.

However, while object allocations produce a direct and easy to understand performance impact, the costs of garbage collections are easily overlooked. Programmers are often unaware of the proportion their application spends on collecting garbage. They often also do not know that the lifetime and the modification patterns of objects can have a big influence on GC behavior. This unawareness may result in bad throughput, long response times, or even in applications that are completely unresponsive, due to long GC pauses.

Managed environments such as the Java virtual machine (VM), which we used in our research, often provide hundreds of parameters for tuning the garbage collector to the needs of a specific application. Each of these parameters comes with a default value that has been selected to fit the 'average application'. However, defining an average application is hard considering today's application diversity. Brecht et al. [8] observed that the default configuration of a garbage collector is rarely perfect for any given application. As only a few parameters come with guidelines on how to choose appropriate values, most operators stick to tuning only this small set of parameters, ignoring others which might lead to additional improvements. Due to the lack of documentation, they often exhaustively profile their application with different GC configurations, having only a faint clue of what they are doing. This attempt is tedious, and might even be futile, due to the sheer number of parameters, the lack of knowledge about the GC implementation, and the unknown influence of each parameter. Thus, operators have to spend a lot of time for tuning their application, often without finding a configuration that provides a significant improvement.

In this paper, we propose to use iterated local search to automatically find an application-tailored GC configuration in a black-box manner. We also present experiments showing that our approach decreases GC time and thus overall run time significantly on well-known Java benchmarks and on a real-world industrial application. Furthermore, we provide explanations on why an optimized configuration is well-suited for the respective application.

Our scientific contributions are a method for automatically tuning a Java garbage collector for a specific application as well as an empirical evaluation for a large set of benchmarks and three widely used garbage collectors.

We conducted our research in cooperation with Compuware Austria GmbH. Compuware develops leading-edge performance monitoring tools for multi-tier Java and .NET applications. In their own applications as well as in applications of their customers, high GC times are a problem that currently cannot be resolved with Compuware's tools.

This paper is structured as follows: Section 2 provides a basic understanding of garbage collection; Section 3 describes the problem we want to address in more detail; Section 4 illustrates our optimization approach; Section 5 describes our research method and experimental setup as well as detailed results; Section 6 discusses related work; Section 7 shows open research questions; and Section 8 concludes the paper.

2. GARBAGE COLLECTION IN HOTSPOT

This section provides a basic understanding of the garbage collectors available in the Java 8 Hotspot^{TM} VM.

The *Serial GC* is the oldest of them and was designed for single-core machines with a small heap. It is a stop-the-world collector, meaning that it suspends the entire VM during garbage collection. Furthermore, it is a generational collector [16, 24], i.e., it divides the heap into a young generation and an old generation of objects. The young generation consists of a *nursery* and two survivor spaces, called *from space* and *to space*. New objects are allocated in the nursery. When the nursery is full, i.e., when there is not enough free space for a new object, all live objects are marked recursively (mark phase) based on the root pointers (i.e., static variables, local variables and pointers originating from other heap spaces). Subsequently, all marked objects in the nursery and in the from space are copied to the to space, and the two survivor spaces are swapped, resulting in an empty nursery and an empty to space (copy phase). Copying collectors waste memory because one survivor space is always empty, but they allow for fast collections because the collection time only depends on the number of live objects and not on the amount of garbage. When an object has survived a certain number of garbage collections in the young generation, it is tenured, i.e., it is copied into the old generation. If the old generation becomes full, it must be garbage collected as well. For the old generation, the Serial GC uses a mark-and-compact scheme. First, all live objects are marked; then all marked objects are moved towards the beginning of the heap, while all pointers to them are adjusted. Mark-and-compact collection is significantly slower than copying collection but it does not waste memory for empty semi-spaces. Furthermore, collections of the young generation (minor collections) are done much more frequently than collections of the old generation (major collections) because most objects die young, and thus the old generation does not fill up so quickly.

The *Parallel GC* uses the same heap layout and the same algorithms as the Serial GC. However, each phase is done in parallel by multiple threads, decreasing the garbage collection time considerably on multi-core processors.

The *Concurrent Mark and Sweep GC* is again generational and is based on the Parallel GC. However, it is not a stop-the-world collector. Rather, it reduces the time of major collections by doing parts of its work (e.g., marking) concurrently in the background while the application (the mutator) runs in the foreground and might even modify references, thus interfering with the collector. Using the Concurrent Mark and Sweep GC increases application responsiveness, especially if all available cores are used by the mutator. On the other hand, if the mutator is under heavy load and thus interferes with the collector heavily, the collector might have to revisit parts of the heap because references were modified by the mutator.

The *Garbage First GC* [10] is a generational collector. It divides the heap into a number of small regions; the young generation (i.e., the nursery and the survivor spaces) and the old generation are logical sets of such regions and are not contiguous. The marking phase is done concurrently, similar to the Concurrent Mark and Sweep GC, but regions with only a few live objects are collected first in order to free as much memory as possible per collection. Thus, this collector can deal with large heaps efficiently, because long collections of many regions arise rarely.

3. PROBLEM

The Hotspot^{TM} VM [20] comes with 681 parameters (1338 with a debug build), most of them documented only by sparse comments in the VM's source code. These parameters are only exposed with the `PrintFlagsFinal` VM flag.

To get an overview of this mass of parameters, we categorized them into several groups, e.g., compiler parameters, memory parameters, or threading parameters. As our goal was to parameterize the garbage collector, we focused on the memory group. This group was again split into one subgroup for each garbage collector of the Hotspot^{TM} VM. We also introduced an additional group of parameters affecting all garbage collectors, e.g., parameters setting the field layout of an object or the size of allocation buffers. Every group is stripped from all tracing and debugging flags so that only performance-relevant parameters remain.

Table 1 shows all subgroups of the memory group and their respective sizes.

Group	Parameters
Generic Memory	17
Parallel GC (Parallel Old GC)	37
Garbage First GC (G1)	45
Concurrent Mark and Sweep (CMS)	103
Parallel New GC (ParNew)	41
Serial GC	37
	280

Table 1: Memory parameter groups

Some parameters, such as the preferred heap size and just-in-time compiler, are chosen automatically at startup by the virtual machine. This mechanism, called *Ergonomics* [17], takes the underlying hardware (e.g., the number of processors) as well as a pause-time goal into account to choose values for a small set of parameters automatically. For example, the parameter `ParallelGCThreads` is automatically set to the number of available processors when using a parallel stop-the-world collector. Unfortunately, this mechanism takes only static information into account and cannot adjust parameters in response to program characteristics.

Furthermore, parameters are often added or removed from one VM release to the next, making previous tuning results obsolete. If an application is executed on different VMs, the GC parameters might differ entirely. Tuning GC parameters automatically counteracts these problems in addition to improving performance.

4. APPROACH

Our approach is to use iterated local search for tuning a garbage collector to the specific needs of a given application.

The core of this method is an optimization algorithm that adjusts parameter values so that the performance is maximized. For the optimization, we need a parameter model describing the available parameters and their respective legal values, as well as an objective function able to evaluate a given configuration by returning a single value describing the induced performance. This function will profile the application with the given parameter values because it cannot compute or guess the induced performance in advance. Figure 4 shows the individual elements of our approach.

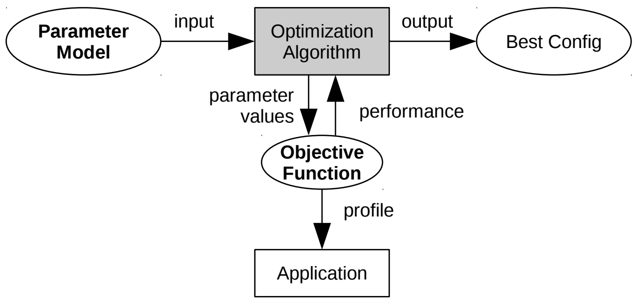

Figure 1: Approach

For the optimization algorithm we decided to use ParamILS [14], an existing optimization framework that is publicly available and easy to use. In the future, however, we also plan to try other optimization frameworks.

4.1 Parameter Model

The input of the optimization algorithm defines the parameter model, with the names of all available parameters, their valid values, and their start values. Furthermore, it includes a description of the relationships between parameters, e.g., parameter A must not be set if parameter B has a certain value. Figure 2 shows a partial model of four GC parameters in ParamILS-specific syntax.

```
XX:TargetSurvivorRatio {25, 50, 75} [50]
XX:?DisableExplicitSystemGC {0, 1} [0]
XX:?UseAdaptiveSizePolicy {0, 1} [0]
XX:?UseAdaptiveSizePolicyWithSystemGC {0,
    1} [0]

Conditionals:
XX:?UseAdaptiveSizePolicyWithSystemGC |
    XX:?DisableExplicitSystemGC in {0}
XX:?UseAdaptiveSizePolicyWithSystemGC |
    XX:?UseAdaptiveSizePolicy in {1}
```

Figure 2: ParamILS input example

The parameter `XX:TargetSurvivorRatio` is a numeric parameter with the legal values 25, 50, and 75, and a start value of 50. The parameters `XX:?DisableExplicitSystemGC`, `XX:?UseAdaptiveSizePolicy`, and `XX:?UseAdaptiveSizePolicyWithSystemGC` are boolean parameters, 0 being false and 1 being true. As the names suggest, `UseAdaptiveSizePolicy` enables an adaptive heap-resizing policy during garbage collection. `XX:?UseAdaptiveSizePolicyWithSystemGC` enables this policy also when `System.gc()` is called, whereas `System.gc()` calls can be disabled by setting `XX:?DisableExplicitSystemGC` to true. Obviously, allowing adaptive resizing with `System.gc()` makes no sense if adaptive resizing was disabled in the first place. Such constraints are defined in the last section of the input file, headed by the `Conditionals` keyword. Each line in this section describes that the first parameter is only to be set if the value of the second parameter is within the specified values. This can reduce the search space for the optimization algorithm significantly, producing better results faster.

4.2 Objective Function

In addition to the parameter model, we need an objective function translating given parameter values into a performance metric. In our case, the objective function is implemented by a script that starts a Java application several times and extracts its garbage collection time via *Java Management Beans* and returns the median to the optimization algorithm. As we extract the aggregated garbage collection time, the optimization algorithm will optimize for overall throughput. Other optimization goals that could be considered will be briefly discussed in Section 7.

4.3 ParamILS

When provided with a parameter description (i.e., a parameter model) and an appropriate problem launcher (i.e., an objective function), ParamILS can optimize the parameter settings for any kind of problem. For finding an optimum, the solution space (i.e., all sets of parameter values) is searched using an iterated local search plus a heuristic for making random changes to the configuration from time to time in order to avoid getting stuck in local minima. To determine the quality of a configuration, the quality metric value returned by the objective function is used. This approach is very similar to *hill climbing*, i.e., it changes one parameter at a time until no more improvement is observed, then it repeats the same with the next parameter, and so on. Furthermore, ParamILS introduces a technique called adaptive capping, which aborts runs as soon as they become obvious to not yield any improvement. The optimization algorithm and its implementation are explained in more detail in Hutter et al. [14].

5. EXPERIMENTS

We used our approach described in Section 4 to find the best GC parameter settings for several Java benchmarks using an iterated local search algorithm that tunes parameter values in order to find the smallest overall GC time for a given application and a given input. This section describes our experiments and their results for three different GCs.

5.1 Setup and Research Method

Figure 3 shows the setup of our experiment. As described in Section 4, we use ParamILS as a configuration optimizer. The output of the optimizer, i.e., the best configuration found, is piped to the configuration minimizer, which eliminates all parameters that retained their default values. This minimum configuration is used by the validator to execute several runs, both with the default configuration and with the minimum configuration, to make more detailed quality measurements.

Since the number of possible parameter configurations is huge, the optimizer cannot explore them all. So we have to stop after a certain number of runs or after a certain time. We decided to stop the optimization of an application after

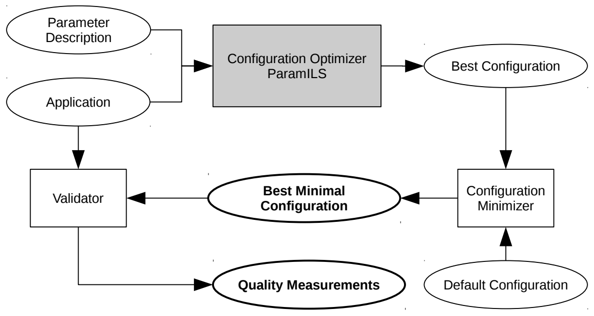

Figure 3: Experiment setup

4 hours. Experiments showed that not much improvement is to be expected after that time for most of our benchmarks, but of course different termination criteria could be used for other experiments. Our experiments optimize for throughput, i.e., the quality metric to be minimized is the overall GC time in an application. Other optimization objectives are discussed in Section 7.

The following subsections describe our research method in more detail, i.e., the selection of benchmarks, the selection of garbage collectors, and the definition of quality metrics.

5.1.1 Benchmarks

To get a broad set of benchmarks, we used applications from various sources. We looked at the DaCapo 2009 benchmark suite, based on Blackburn et al. [7, 1], and selected seven GC-intensive benchmarks, i.e., eclipse, h2, jython, sunflow, tomcat, tradesoap and xalan. Unfortunately, eclipse crashes on Java 8 [20] and was therefore excluded from our selection. In addition to that, we selected the SPECjbb 2005 benchmark which puts a lot of pressure on the garbage collector. The DaCapo benchmarks were always executed with the largest input supported and the SPECjbb benchmark was executed with eight warehouses. For each of these benchmarks, we experimentally determined the minimum heap required for execution. Table 2 shows our benchmarks as well as their minimum heap sizes and average run times when using Java 8 and the Parallel GC.

Benchmark	**Min. Heap [MB]**	**Run Time [s]**
DaCapo h2	300	36.89
DaCapo jython	40	11.03
DaCapo sunflow	10	6.49
DaCapo tomcat	75	6.08
DaCapo tradesoap	25	18.18
DaCapo xalan	10	11.00
SPECjbb	300	451.68

Table 2: Benchmarks used for the experiments

5.1.2 Garbage Collectors

To show that our approach is applicable to any garbage collector, we conducted our experiments with three different garbage collectors and their respective parameters:

- The *Parallel GC* is the default garbage collector, making it one of the most frequently used GCs.

- The *Concurrent Mark and Sweep GC* was selected because the customers of our industrial partner make heavy use of it.

- The *Garbage First GC* (also called G1 GC) was included because is uses a relatively new algorithm, and is therefore quite different from the other garbage collectors. Furthermore, it enjoys increasing popularity with large server applications.

We excluded parameters setting the heap size because there is ample evidence (e.g., Yang et al. [27], Brecht et al. [8]) that the heap size has a significant impact on garbage collector performance. Furthermore, if we enable the optimizer to adjust the heap size as well, it will always choose the biggest allowed value.

5.1.3 Quality Metrics

We defined four quality metrics for determining the quality of a garbage collector configuration:

- The *garbage collection time* is the overall time the benchmark spent on collecting garbage. Minimizing this metric is the main goal of our optimization.

- The *run time* or *throughput* determines the impact of the optimized configuration on the overall application behavior. The DaCapo benchmarks process a given input and terminate subsequently, making the run time a good performance indicator. The SPECjbb benchmark, on the other hand, runs for a fixed amount of time, making the run time a useless metric. In this case, we rather measure the throughput.

- The *garbage collection frequency* describes the number of garbage collection cycles that occurred during the benchmark execution. Together with the garbage collection time, this metric can be used to estimate the average length of garbage collection pauses.

- The *peak heap usage* indicates the maximum amount of live memory during benchmark execution.

Whenever these metrics were measured, we executed a number of warm-ups first in order to stabilize the caches and to JIT-compile all the hot spots.

We examined all optimized configurations and their induced behavior by injecting custom agents into the VM, which extract information such as the GC frequencies and the GC times. These agents use the *Java Virtual Machine Tool Interface* (JVMTI) to access VM-internal information.

In order to better understand the results and offer detailed interpretations, we also used the built-in GC logging mechanism as well as VM instrumentation to collect additional data, e.g., the run time of individual GC phases or the average object ages. Our custom VM does not introduce any costly computations, but rather aggregates and exposes already existing information. For example, the average object age is computed during the marking phase, because the GC has to traverse all objects anyway. Nevertheless, to reduce the risk of tainted results, all figures in the following subsections have been created without GC logging and with an unmodified VM.

5.1.4 Hardware and Software

We ran our experiments on an Intel® Core$^{\text{TM}}$ i7-3770 CPU @ 3.40GHz×4 (8 Threads) on 64-bit with 18GB RAM running Ubuntu 12.10 Quantal Quetzal with the Kernel Linux 3.5.0-38-generic. All unnecessary services were disabled and the experiments were always executed in text-only mode. We used the OpenJDK 8 Early Access Release b100 [20], because significant changes were made to the garbage collectors compared to Java 7, e.g., the permanent generation was removed.

5.2 Results

This subsection provides an overview of the results of our experiments. A detailed discussion will follow in Section 5.3.

Figures 4 - 7 show the measured quality metrics for all benchmarks with a specific garbage collector. Each subfigure is a histogram with 2 bars per benchmark; the left (dark) bars indicate the results of the default parameter configurations; the right (light) bars show the results of the optimized parameter configurations. All values are medians of multiple runs, normalized with respect to the value of the default configuration. We used the median because it is more realistic than the peak performance, more stable with respect to outliers than the arithmetic mean, and more meaningful for our metrics than the geometric mean. The error interval on top of each bar indicates the standard deviation. Due to the normalization, a large error interval on a long running benchmark might indicate the same standard deviation as a smaller error interval in a shorter running benchmark (cf. Table 2). For benchmarks that execute for a fixed amount of time, i.e., SPECjbb, the throughput is normalized by dividing the default configuration throughput by the optimized throughput. The rightmost two bars represent the arithmetic mean of the individual speedups.

Figure 4 shows the results of the parameter optimization for the Parallel GC without any memory pressure applied, i.e., the VM is allowed to increase the heap size arbitrarily. Please note that the heap usage diagram has a different scale and that the heap size is not explicitly set but is a result of the optimized configuration. The results show that the overall run time is decreased by 9% in the tradesoap benchmark and shows slight speedups for almost all other benchmarks. Of course, the impact on the overall run time depends on how much time a benchmark spends for garbage collection. The GC time, on the other hand, has been reduced significantly for all benchmarks. With the optimized parameter configuration, the tradesoap benchmark does not need any garbage collection at all, reducing the GC frequency and the GC time to zero. Sure enough, these results are mostly due to the fact that the heap size has been increased by a factor of 13. Large heaps lead to less garbage collections, because it takes longer to fill the heap. Furthermore, the GC time depends only on the number of live objects which is independent of the heap size.

To show that the GC time depends on more than just the heap size and that it can be reduced by optimizing the GC configuration, we decided to artificially apply memory pressure to all benchmarks. This was done by experimentally determining the minimum heap size for a benchmark and setting the actual heap size to a multiple of that value. Table 2 shows the determined minimum heap sizes of our benchmarks. For the following experiments, we decided to use a maximum heap size that is twice the minimum heap size for every benchmark because this seems to create a realistic memory pressure. When the memory pressure is decreased, i.e., the heap is increased, measurements have shown that the results converge to the values shown in Figure 4.

5.2.1 Parallel GC

Figure 5 shows the results of optimizing the Parallel GC with twice the minimum heap size. Due to the memory pressure, the heap cannot grow arbitrarily but the memory manager must get along with the space available. In some cases, i.e., in h2, jython, and tomcat, the heap usage is higher than with the default configuration, although it is below the allowed maximum heap size. Thus, the optimized configuration obviously uses the available space more efficiently than the default configuration, e.g., by choosing better sizes for the survivor spaces. The GC time and the GC frequency have both been reduced on all benchmarks. Compared to the results without memory pressure (Figure 4) the overall run time speedup is significantly higher (by up to 42% for xalan) because the garbage collection ratio is bigger, i.e., due to memory pressure the application spends a larger percentage of its run time on garbage collection. The h2 and the SPECjbb benchmarks show less improvement than the others, because due to their larger execution time, fewer runs could be executed by the optimizer in the fixed time frame.

5.2.2 Concurrent Mark and Sweep GC

The results of optimizing the Concurrent Mark and Sweep GC (Figure 6) show only small improvements, indicating that the default parameter settings were more or less adequate for our benchmarks. One might also question the statistical significance of the results considering the small speedups and their standard deviations.

5.2.3 Garbage First GC

Optimizing the Garbage First GC yields results (Figure 7) that are similar to those of the Parallel GC. Some benchmarks show a remarkable reduction in GC time, which is, however, not observable in the overall run time, because GC time seems to be only a small fraction of the run time in this configuration. Nevertheless, the overall run time of SPECjbb could be reduced by 10%. The sunflow benchmark uses only a fraction of the heap space available with the default configuration. The optimized configuration utilizes the available space better, resulting in a spike in the heap usage. Furthermore, the standard deviation is much higher compared to the Parallel GC. This indicates that, due to the large number of small heap regions, the Garbage First GC is easily influenced by external factors, such as the object allocation order or the heap layout. The optimization on tradesoap does not find a configuration resulting in a significant speedup, indicating that the default configuration is already well suited for this application. The GC frequency of h2 increased whereas the GC time dropped, meaning that although the application ended up collecting garbage more often, the individual GC pauses where shorter, resulting in an overall reduction of GC time.

5.3 Detailed Results and their Interpretation

We have selected two benchmarks for which we will discuss the results in more detail. These benchmarks have been selected based on their differing optimum configurations.

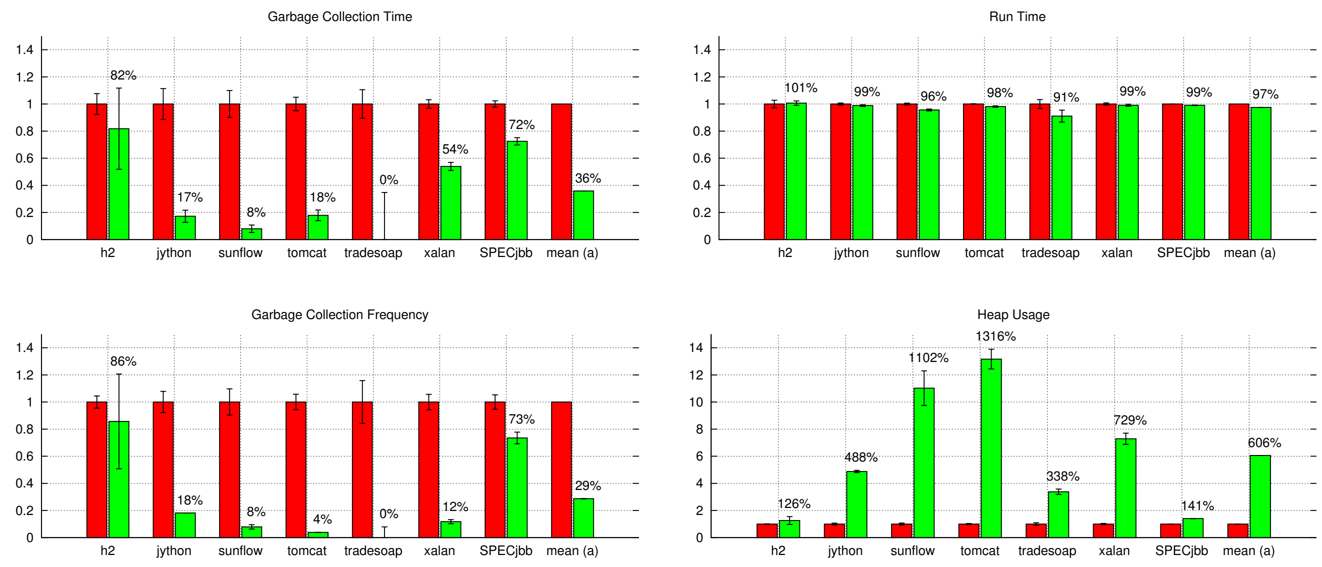

Figure 4: Optimization results for the Parallel GC (normalized, without memory pressure)

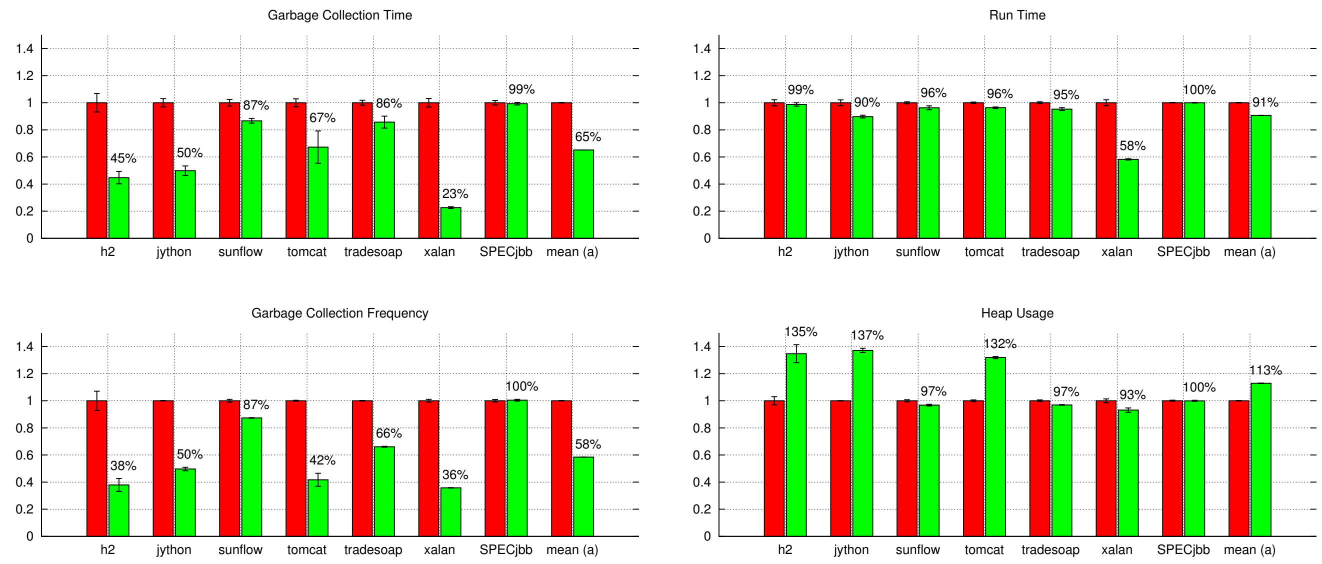

Figure 5: Optimization results for the Parallel GC (normalized, with memory pressure)

5.3.1 Parallel GC applied to xalan

This section looks at the optimization of the Parallel GC when running the DaCapo xalan benchmark. Figure 8 shows the GC frequencies, the GC times, and the heap usage for the xalan benchmark broken down into different heap subspaces and their respective GC algorithms. The left (dark) bars are the values produced by the default configuration, and the right (light) bars are produced by the optimized configuration. *Scavenge* denotes the collection of the young generation and *MarkSweep* the collection of the old generation. *Eden Space* and *Survivor Space* together make up the young generation (the second survivor space required by the scavenge algorithm is not shown as it is always empty). Please note that this figure shows the peak space usage and not the actual space size. However, the peak usage is equal to the size in the eden space and the old generation because GCs mostly occur when a space is full. The size of the survivor space depends on the usage and the value of the *TargetSurvivorRatio* parameter. The default value for this parameter is 50, i.e., the survivor space is sized so that up to 50% are occupied after a minor GC, meaning that the size of the survivor space is approximately twice the usage. Figure 8 shows that both the GC frequency and the GC time dropped dramatically in the optimized configuration when more space was given to the young generation. If the young generation is larger it needs less frequent collections and thus gives young objects more time to die between collections. If objects die before they are tenured, the old generation be-

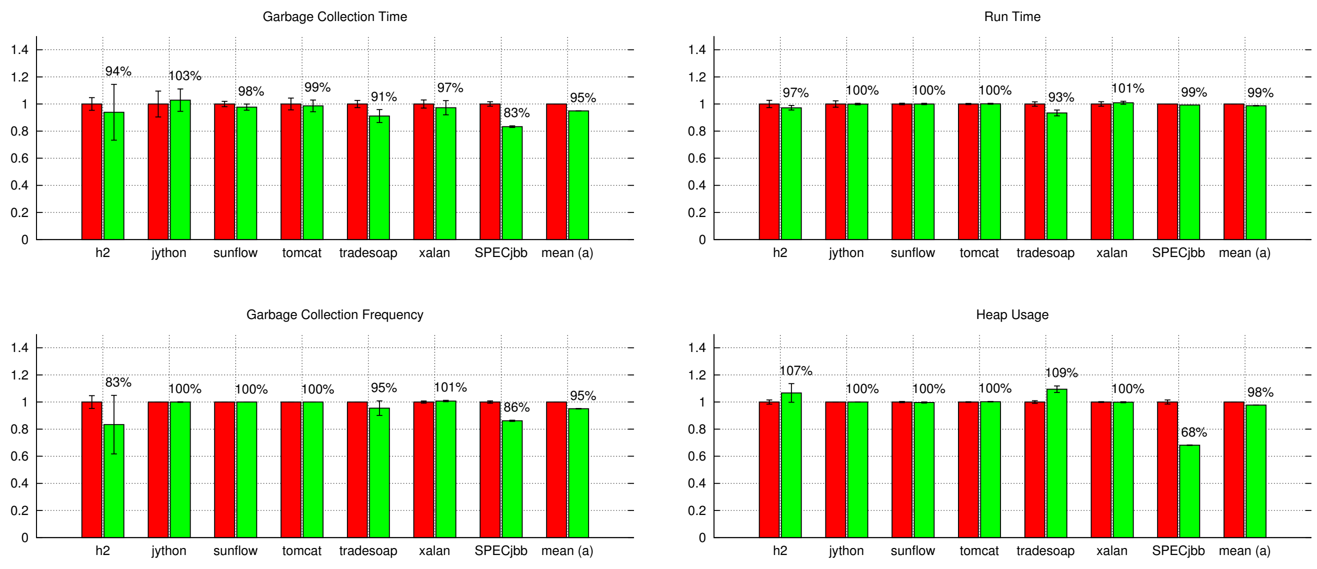

Figure 6: Optimization results for the CMS GC (normalized, with memory pressure)

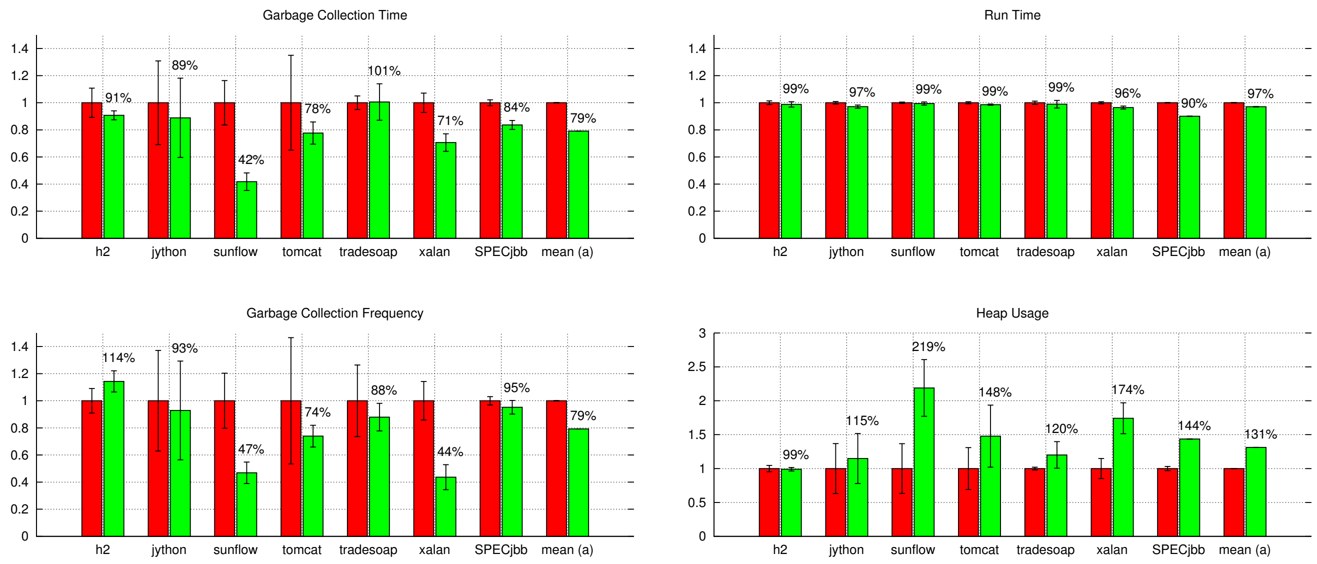

Figure 7: Optimization results for the G1 GC (normalized, with memory pressure)

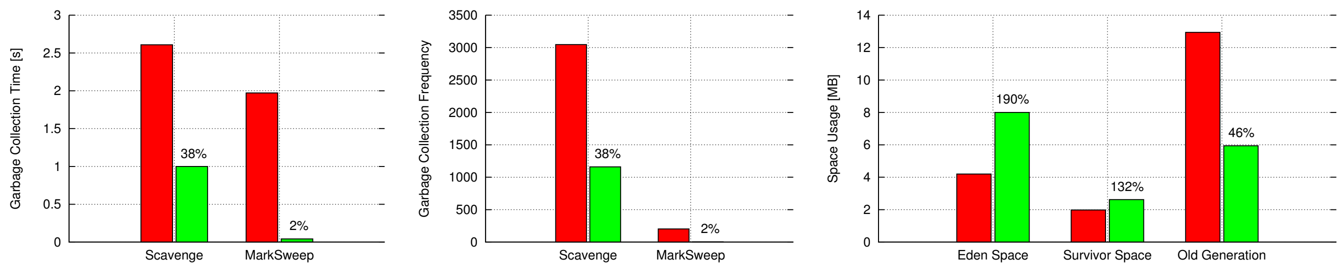

Figure 8: Detailed optimization results for xalan (Parallel GC with memory pressure)

comes smaller and the expensive *MarkSweep* GC can run less frequently. Obviously, the xalan benchmark has many short-living objects ([7]) so that the optimized configuration is beneficial here.

Figure 9 shows that with the optimized configuration every *Scavenge* run frees more than four times as much memory as with the default configuration. This confirms our conjecture that in the optimized configuration most objects die before the next GC run.

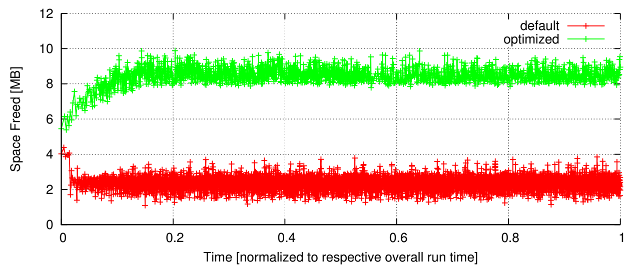

Figure 9: Amount of memory freed in the young generation for xalan (Parallel GC with memory pressure)

The optimized configuration and its interpretation.

Table 3 compares the optimized (new) parameter values with their default (old) values for the xalan benchmark. Parameters that retained their default values are excluded.

Parameter	old	new
AdaptiveSizeDecrementScaleFactor	4	2
AdaptiveSizeMajorGCDecayTimeScale	10	5
AdaptiveSizePolicyCollectionCostMargin	50	40
AdaptiveSizeThroughPutPolicy	0	1
BindGCTaskThreadsToCPUs	-	+
CollectGen0First	-	+
MinHeapFreeRatio	40	20
MinSurvivorRatio	3	1
NewRatio	2	1
OldPLABSize	1024	2048
PLABWeight	75	80
ResizeOldPLAB	+	-
SurvivorPadding	3	2
TargetSurvivorRatio	50	70
TenuredGenerationSizeIncrement	20	30
TenuredGenerationSizeSupplement	80	85
TenuredGenerationSizeSupplementDecay	2	16
UseAdaptiveGCBoundary	-	+
UseAdaptiveGen.SizePolicyAtMajor	+	-
UseAdaptiveGen.SizePolicyAtMinor	+	-
UseAdaptiveSizeDecayMajorGCCost	+	-
UseAdaptiveSizePolicyFootprintGoal	+	-
YoungGenerationSizeIncrement	20	30
YoungPLABSize	4096	1024
YoungGenerationSizeSupplement	80	75
YoungGenerationSizeSupplementDecay	8	2

Table 3: Optimized parameter configuration for xalan (Parallel GC with memory pressure)

The *NewRatio* parameter configures the size ratio between the old generation and the young generation, i.e., a default value of 2 results in an old generation that is twice as big as the young generation. Changing this value to 1 doubles the size of the young generation, thus cutting the GC frequency of the young generation at least in half. Enabling the *UseAdaptiveGCBoundary* flag allows the VM to move the boundary between heap spaces, making it easier to further increase the size of the young generation. Decreasing the *MinSurvivorRatio* from 3 (i.e., the eden space is at least three times as big as a single survivor space) to 1 results in bigger survivor spaces. Therefore, more objects can be kept in the young generation for a longer time before they are promoted to the old generation. Additionally, increasing the *TargetSurvivorRatio* enables a bigger fraction of the survivor space to be occupied and thus allows more live objects in the survivor spaces without forcing a premature promotion. *CollectGen0First* forces a collection of the young generation just before collecting the old generation, avoiding a major GC immediately followed by a minor GC. Furthermore, the *YoungPLABSize* (i.e., the size of the young-to-old promotion buffer) is decreased because it is hardly used in xalan. The *OldPLABSize* (i.e., the size of the promotion buffer used for compacting objects in the old generation), is increased to reduce the frequency of buffer overflows during major GCs. Furthermore, the optimized configuration disables several adaptive policies (e.g., *UseAdaptiveGenerationSizePolicyAtMinor*, *UseAdaptiveGenerationSizePolicyAtMajor*, and *UseAdaptiveSizePolicyFootprintGoal*) that would interfere with other chosen parameter values. Please note that these dependencies were not in our parameter model, but were found automatically during optimization. The optimized configuration also contains some false positives, such as *AdaptiveSizeThroughPutPolicy*, *AdaptiveSizeDecrementScaleFactor*, and *AdaptiveSizeMajorGCDecayTimeScale*. These parameters were modified by the optimization algorithm before it disabled the corresponding policies. There was not enough time for the optimization algorithm to determine that these parameters are now without any impact.

Note that this parameter configuration is specific for the xalan benchmark and would not necessarily produce good results for other benchmarks.

Optimization.

Figure 10 shows the progress when optimizing the parameter configuration for xalan. The horizontal axis represents

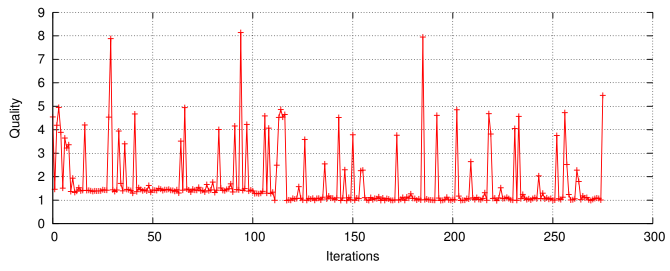

Figure 10: Optimization progress for xalan (Parallel GC with memory pressure)

the number of iterations, the vertical axis represents the quality of the respective iterations (i.e., GC time in seconds). As expected, a relatively good configuration is found very early. Afterwards, the algorithm tries to further optimize the configuration, occasionally making a bad choice resulting in spikes, but returning immediately if a change does not show an improvement. In all our benchmarks, a good configuration was already found after a few iterations, and no further significant improvements were made after a total optimization time of 4 hours.

5.3.2 Garbage First GC applied to SPECjbb

This section looks at the optimization of the Garbage First GC when running the SPECjbb benchmark. In terms of allocation rates and absolute GC times, this benchmark puts significantly more pressure on the garbage collector than the xalan benchmark. Moreover, measurements have shown that, although many objects die young, the average object age at death is considerably higher than in xalan. Figure 11 shows the GC time and the GC frequency per generation as well as the heap space usages. It turned out that the default parameter configuration kept almost all objects in the young generation. However, tenuring long-living objects earlier would decrease the GC time of the young generation at the expense of the old generation. The optimized configuration has found the sweet spot, or at least an approximation of it, for the object distribution between generations.

The optimized configuration and its interpretation.

Table 4 compares the optimized parameter values with their default values for the SPECjbb benchmark. Values marked with a '*' are chosen at start-up time for our machine by the GC ergonomics.

Parameter	old	new
G1ConfidencePercent	50	40
G1ConcRefinementGreenZone	8*	4
G1ConcRefinementServiceIntervalMillis	300	150
G1ConcRefinementYellowZone	24*	32
G1ConcMarkStepDurationMillis	10	5
G1ConcRSHotCardLimit	4	16
G1HeapRegionSize	1MB*	4MB
G1HeapWastePercent	10	15
G1MixedGCCountTarget	8	16
G1ReservePercent	10	5
G1RSetUpdatingPauseTimePercent	10	15
G1RSetScanBlockSize	64	128
G1SATBBufferEnqueueThresholdPercent	60	30
G1UseAdaptiveConcRefinement	+	-

Table 4: Optimized parameter configuration for SPECjbb (G1 GC with memory pressure)

The parameter *G1RSetUpdatingPauseTimePercent* defines a limit for the time used updating the remembered sets. This limit affects the concurrent refinement and when a GC is triggered. These sets contain all root pointers per heap region, i.e., all pointers into the region originating from other regions. Increasing this parameter enables the garbage collector to process more updates to this set necessary for the heavy load of the SPECjbb benchmark. Increasing the *G1ConcRSHotCardLimit* allows more pointer updates in a memory card before the card is considered hot, thus triggering a garbage collection later than usual. If code with many pointer updates but only few allocations is executed, a higher limit prevents unnecessary collections. The *G1ConcRefinementGreenZone* and *G1ConcRefinementYellowZone* define how many update buffers for the remembered set will be left in the queue (see Detlefs et al. [9]) and at which queue size how many concurrent refinement threads are triggered. Reducing the green zone results in fewer buffers left in the queue whereas increasing the yellow zone results in concurrent processing being triggered later as usual. Therefore, less concurrent refinement threads are triggered less often but have to process a bigger queue, reducing the interference between the concurrent refinement and the mutator. This behavior is favorable for heavy load with many pointer updates, as observed in the SPECjbb benchmark. The *G1ConcMarkStepDurationMillis* is decreased, resulting in smaller incremental steps in the marking phase. Thus, if many pointers are modified (as it is the case with SPECjbb) the necessary re-marking steps due to the mutator interference are also shorter, decreasing the time spent on remarking and thus the overall GC time. To reduce parallelization overhead, the *G1RSetScanBlockSize* is increased, resulting in bigger chunks for each worker thread. An increased *G1SATBBufferEnqueueThresholdPercent* leads to more SATB (Snapshot At The Beginning) buffers to be enqueued, enabling the garbage collector to handle pointer updates more quickly. Finally, the *G1HeapRegionSize* is increased, resulting in improved locality of sequentially allocated objects (i.e., it is more likely that sequentially allocated objects are in the same heap space) and in longer intervals between garbage collections. Therefore, objects have more time to die and each garbage collection can free a bigger fraction of a heap region.

Similarly to xalan, this parameter configuration was optimized to fit the exact needs of the SPECjbb benchmark.

5.4 Real-world Experiment

In addition to the DaCapo and SPECjbb benchmarks, we conducted our experiments also on a real-world industrial application. This subsection describes the modified setup and the results of this experiment.

5.4.1 Modified Setup and Research Method

The application for which we tuned the garbage collector is the dynaTrace Server 5.5 from our industrial partner Compuware. The dynaTrace Server receives monitoring information, such as stack traces, allocation events, garbage collector information, and captured parameter values, from agents that are injected into other real-world applications in order to monitor them. This information has to be processed and aggregated in real time and has to be stored into a performance warehouse (i.e., a database) to be accessible via the dynaTrace Client.

When the load is too high, i.e., when the agents send too much data per time unit to the server, it starts to ignore incoming data. Compuware has experimentally determined the maximum data rate that the server is able to handle without having to skip anything for a given hardware setting and given VM parameters. For our experiment, we used 68 agents to send data at that rate, resulting in a server that is used up to its maximum capacity but is still handling the received data correctly.

Due to the typical environment characteristics of Compuware's customers, the heap size was fixed at 8GB. Until now, Compuware achieved the best performance with the Concurrent Mark and Sweep GC. Therefore, we optimized the configuration of this GC.

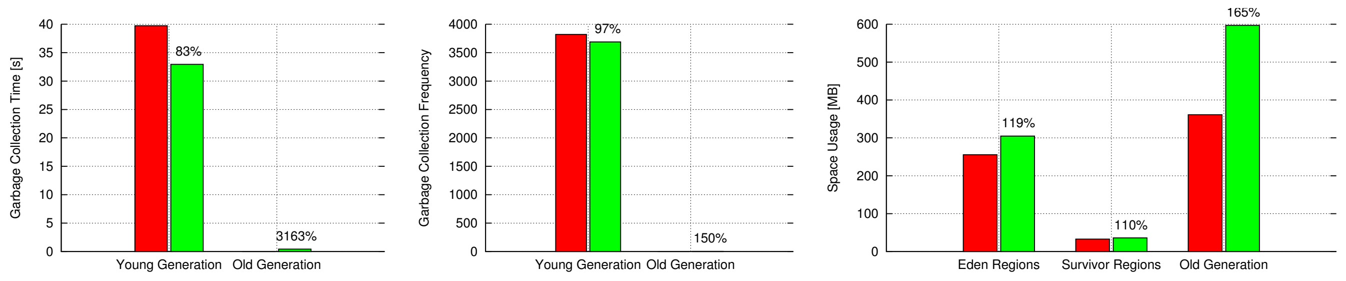

Figure 11: Detailed optimization results for SPECjbb (G1 GC with memory pressure)

As the server is handling requests continuously, we fixed a single optimization iteration, i.e., a single run with a given configuration, to 15 minutes. The optimization was conducted for 2.5 days, resulting in about 600 iterations.

Similar to the other benchmarks, the objective function was defined as the aggregated GC time during 15 minutes under load. We had to add additional safeguards to the objective function because some configurations, while leading to an overall GC time improvement, had significantly longer GC pauses. Such configurations resulted in a decreased responsiveness, forcing the server to skip data. Thus, the objective function continuously reads the server logs and reports a configuration as crashed when the server starts skipping data or when it loses the connection to the agents due to long GC pauses. Such effects arise early, which enables the optimization algorithm to save time because it can prematurely abort runs with such configurations.

The server ran on 4 × MJ3GK E7540 @ 2.4GHz×6 Intel® Xeon® Multi Core Dunnington D0 on 64-bit with 96GB RAM running Linux Ubuntu 10.10 Maverick Meerkat. The agents creating the load ran on separate machines in the same network in order to avoid biased results.

5.4.2 *Results*

Figure 12 shows the aggregated GC time and the GC frequency before (left, dark) and after (right, light) the optimization. Although the GC frequency increased up to 161%, the GC time dropped to 58%. We do not show the run time or the throughput because the agents were configured to send requests at the same fixed rate before and after the optimization.

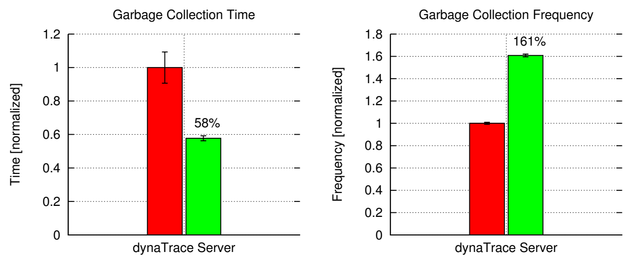

Figure 12: Optimization results for the dynaTrace Server (normalized, with the CMS GC)

The detailed results (Figure 13) show that the eden space (*Par Eden Space*) has been cut in half, resulting in an increased minor collection frequency, whereas the old generation space (*CMS Old Gen*) was increased by 26%. Most objects allocated by the dynaTrace Server are temporary objects that are created during request processing. These objects usually die before the first GC cycle. Therefore, 73% of the GC time was spent on minor collections (*ParNew*). However, there are also long-living objects which the dynaTrace Server keeps in memory to enable fast access for dynaTrace Clients. As these objects are usually stored in caches, they can be collected within the first few major collections. Increasing the old generation space gives them more time to die and thus speeds up collections in the old generation.

The optimized configuration leads to a better balance of short-living and long-living objects between the two generations. As most objects die before the first collection, the size of the young generation can be safely reduced and objects that survive the first collection are promoted to the old generation. Increasing the size of the old generation reduces the major GC frequency and gives long-living objects more time to die there before the next major collection. Preliminary tests have shown that, in this scenario, the transaction throughput can be increased by as much as 14%.

These results show that our approach of automatically tuning GC parameters not only works for smaller benchmarks but also for real-world industrial applications

5.5 Threats to Validity

Impact of individual parameters Although we examined the GC logs carefully and performed additional runs with our instrumented VM in order to be able to explain the optimization effects, more experiments would be required to better understand the impact of each parameter in isolation as well as in combination with others.

Applicability to other VMs Our approach is applicable to any VM that exposes parameters to control garbage collection behavior. In order to verify this, experiments would have to be conducted for other VMs. However, we are confident that our approach yields similar results because they use similar garbage collection algorithms and parameters.

Hardware diversity All benchmarks were executed on the same machine. Although we expect most parameters to be hardware independent, additional experiments would be required to verify this assumption and to analyze potentially different result configurations for the same benchmark.

6. RELATED WORK

To the best of our knowledge, no work has been published on automatically tuning a GC considering *all* parameters. There is some research, though, on choosing proper sizes for

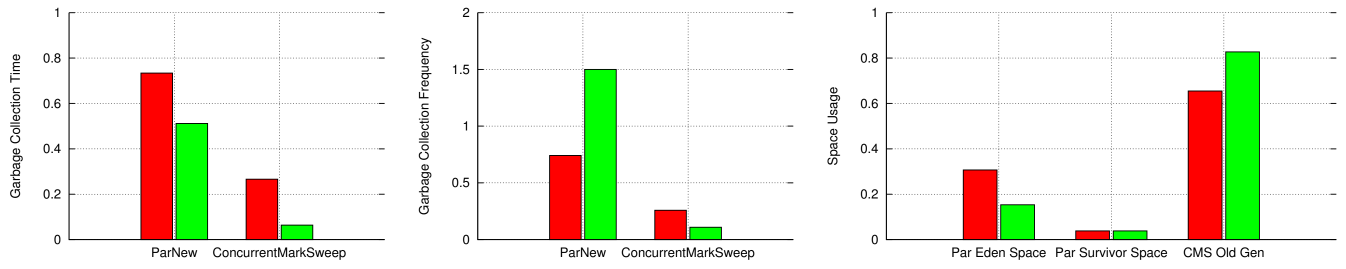

Figure 13: Detailed optimization results for the dynaTrace Server (normalized, with the CMS GC)

the entire heap or for individual spaces, for manual tuning, GC performance, and parameter value selection in general:

Heap space sizes Yang et al. [27] implemented a heap size analysis to minimize paging while maximizing throughput by adjusting the heap size accordingly. Only moderate changes to the garbage collector are necessary to employ their approach, which can reduce the GC time considerably. Brecht et al. [8] experimentally examined the performance impact (e.g., the overall run time, GC pause times, and footprint) of changing the heap size for Java applications. Furthermore, they devised a heuristic algorithm to resize the heap with respect to the observed application behavior. Guan et al. [12] investigated the performance effects of different nursery sizing policies. They proposed a hybrid policy for handling different memory pressure scenarios efficiently, enabling server applications to deal with higher workloads. Balsamo et al. [3] used a queuing model for predicting the optimal activation rate, i.e., the GC frequency to minimize the mean response time. Vengerov [26] developed a mathematical analysis to maximize the throughput based on the sizes of the young and the old generation, as well as on the tenuring threshold. He showed that, using his definition, the heap size and the tenuring threshold converge to their optimal values, achieving the optimal throughput. Valesco et al. [25] proposed a method for dynamic reorganization of the heap in a generational collector. They present two techniques for choosing the percentage of reserved space and show that these techniques can reduce the collection time substantially. Singer et al. [22] introduced the allocation curve as a special form of the demand curve from economics as well as the term allocation elasticity to control heap growth.

Manual tuning Gousios et al. [11] examined the impact of GC tuning on Java server applications. Using the results of their experiments, they devised a number of guidelines to tune the Sun and the IBM virtual machines. However, these guidelines refer only to the heap size and to the selection of a garbage collector. Hirt et al. [13] described some parameters of the JRockit VM and the capability of the JRockit Mission Control to provide statistics about heap space usage. The described parameters are mostly about individual heap space sizes and their resizing, and the selection of GC algorithms. Moreover, the interpretation of the provided heap space statistics is left to the user as only little guidance on tuning is provided.

GC performance Blackburn et al. [6] identified key algorithmic features of three GC algorithms and developed a function for expressing their performance costs based on the heap size.

GC selection Singer et al. [21] implemented a method to select a garbage collection algorithm based on a single profiling run, achieving a significant speedup over choosing the default GC each time. However, they focus on choosing the garbage collector only, ignoring the GC parameters.

GC parameters Singer et al. [23] suggest using decision trees to predict a small set of parameters, i.e., the GC algorithm and one of two new-to-old generation ratios. However, they do not consider the entire set of available parameters. Other sources provide guidelines on how to select values for certain parameters of the Java VM. Oracle [19, 18] provides information about some parameters of the Hotspot^{TM} VM and explains their effect on the garbage collector. However, this description is limited to parameters controlling the size of individual heap spaces, the size of the entire heap, the heap growth, and the GC algorithm. Lee [15] offers similar instructions about choosing the heap size. Beckwith [4] provides more insight into the Garbage First collector and its most important parameters [5]. However, some of the suggested parameter values are experimental and most of them remain unexplained.

7. FUTURE WORK

Optimizing other VM parameters As our approach is of a black-box manner, it is not limited to garbage collection, but could also be applied to other VM parameters such as compiler heuristics or threading behavior.

Choosing other optimization frameworks We used ParamILS because it is easy to adapt and publicly available for academic use. However, one might consider other optimization frameworks as well (e.g., Heuristic Lab [2]) in order to check whether they can find better optima.

Changing the optimization goal Our optimization goal was the overall throughput of an application. Therefore, we used the aggregated GC time as an objective function for the optimization algorithm. However, one could also think of other optimization goals such as the average or maximum GC pause time.

8. CONCLUSIONS

In this paper we proposed a technique for the automatic tuning of GC parameters for specific applications using an optimization tool that applies a modified hill climbing approach. We conducted detailed experiments with a variety of GC-intensive benchmarks from the DaCapo benchmark suite and from SPECjbb 2005 as well as with a real-world industrial application (the dynaTrace Server). The experiments were performed for 3 widely used garbage collectors of the Hotspot^{TM} VM.

Our measurements show that for some benchmarks, the GC time can be reduced by up to 77% leading to an overall run-time speedup of up to 42% relative to the default configuration. The average reduction of GC time across all benchmarks was 35% and the average speedup on overall run time was 9% (for the Hotspot^{TM} default GC).

With dozens of GC parameters, which are scarcely documented and hard to understand, the manual tuning of applications is a tedious task which is often guided by trial and error. Automatic tuning can be an attractive alternative that exploits otherwise hidden GC potential based on the characteristics of specific applications.

9. ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Compuware Austria GmbH.

10. REFERENCES

[1] DaCapo. `http://www.dacapobench.org/`, 2013.

[2] M. Affenzeller. Architecture and Design of the HeuristicLab Optimization Environment. In *Advanced Methods and Applications in Computational Intelligence*, pages 197–261, 2014.

[3] S. Balsamo, G.-L. D. Rossi, and A. Marin. Optimisation of Virtual Machine Garbage Collection Policies. In *Proc. of the Intl. Conf. on Analytical and Stochastic Modeling Techniques and Applications*, pages 70–84, 2011.

[4] M. Beckwith. G1: One Garbage Collector to Rule Them All. `http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All`, 2013.

[5] M. Beckwith. Garbage First Garbage Collector Tuning. `http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html`, 2013.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: the Performance Impact of Garbage Collection. In *Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 25–36, 2004.

[7] 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 *Proc. of the Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[8] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. *Trans. Program. Lang. Syst.*, 28(5):908–941, 2006.

[9] D. Detlefs. Concurrent Remembered Set Refinement in Generational Garbage Collection. In *USENIX Java VM Research and Technology Symp.*, 2002.

[10] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-First Garbage Collection. In *Proc. of the Intl. Symp. on Memory Management*, pages 37–48, 2004.

[11] G. Gousios, V. Karakoidas, and D. Spinellis. Tuning Java's Memory Manager for High Performance Server Applications. In *Proc. of the 5th Intl. System Administration and Network Conf.*, pages 69–83, 2006.

[12] X. Guan, W. Srisa-an, and C. Jia. Investigating the Effects of Using Different Nursery Sizing Policies on Performance. In *Proc. of the Intl. Symp. on Memory Management*, pages 59–68, New York, NY, USA, 2009.

[13] M. Hirt and M. Lagergren. *Oracle JRockit: The Definitive Guide*. 2010.

[14] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stutzle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[15] S. Lee. How to Tune Java Garbage Collection. `http://www.cubrid.org/blog/textyle/428187`, 2012.

[16] H. Lieberman and C. Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, 1983.

[17] Oracle. Garbage Collector Ergonomic. `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/gc-ergonomics.html`, 2013.

[18] Oracle. Java HotSpot VM Options. `http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html`, 2013.

[19] Oracle. Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`, 2013.

[20] Oracle. OpenJDK 8 Early Access Release b100. `http://jdk8.java.net/archive/8-b100.html`, 2013.

[21] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent Selection of Application-specific Garbage Collectors. In *Proc. of the Intl. Symp. on Memory Management*, pages 91–102, 2007.

[22] J. Singer, R. E. Jones, G. Brown, and M. Luján. The Economics of Garbage Collection. In *Proc. of the Intl. Symp. on Memory Management*, pages 103–112, 2010.

[23] J. Singer, G. Kovoor, G. Brown, and M. Luján. Garbage collection auto-tuning for java mapreduce on multi-cores. In *Proc. of the Intl. Symp. on Memory Management*, pages 109–118, 2011.

[24] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 157–167, 1984.

[25] J. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic Management of Nursery Space Organization in Generational Collection. In *INTERACT-8, workshop*, pages 33–40, 2004.

[26] D. Vengerov. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *Proc. of the Intl. Symp. on Memory Management*, pages 1–9, 2009.

[27] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic Heap Sizing: Taking Real Memory into Account. In *Proc. of the Intl. Symp. on Memory Management*, pages 61–72, 2004.