# A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008

Philipp Lengauer[1] Verena Bitto[2] Hanspeter Mössenböck[1] Markus Weninger[2]

[1]Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

[2]Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
verena.bitto@jku.at

## ABSTRACT

Benchmark suites are an indispensable part of scientific research to compare different approaches against each another. The diversity of benchmarks is an important asset to evaluate novel approaches for effectiveness and weaknesses. In this paper, we describe the memory characteristics and the GC behavior of commonly used Java benchmarks, i.e., the DaCapo benchmark suite, the DaCapo Scala benchmark suite and the SPECjvm2008 benchmark suite. The paper can serve as a useful guide to select benchmarks in accordance with desired application characteristics on modern virtual machines as well as with different compilers and garbage collectors. It also helps to put results that are based on these benchmarks into perspective. Additionally, we compare Java's current default collector to the G1 GC.

## Categories and Subject Descriptors

[**General and reference**]: Evaluation; [**General and reference**]: Performance

## Keywords

Java, Benchmarks, Memory Behavior, GC Behavior, DaCapo, DaCapo Scala, SPECjvm2008

## 1. INTRODUCTION

Benchmarks are a state-of-the-art method to determine the quality of virtual machines, compiler optimizations, garbage collection algorithms, and monitoring tools in terms of performance. Results of such measurements are used, for example, to argue the superiority of one garbage collection algorithm over another, to demonstrate the benefits of a new optimization technique, or to evaluate the overhead of a new monitoring method.



*ICPE'17, April 22-26, 2017, L'Aquila, Italy*



DOI: http://dx.doi.org/10.1145/3030207.3030211

Considerable effort has been put into building benchmark suites to represent diverse and real-world applications. The most widely used Java benchmark suites are DaCapo[1] introduced by Blackburn et al. [2], DaCapo Scala[2] introduced by Sewe et al. [7], SPECjvm2008[3] analyzed by Shiv et al. [8] and SPECjbb[4]. Unfortunately, those benchmarks either lack a detailed analysis of their comprising applications or their respective analysis is out of date. Since memory management, compilers, and GC algorithms have evolved since the introduction of those benchmarks, their actual behavior in modern systems is undocumented.

However, when reporting performance measurements, the selection of benchmarks is paramount to be able to report descriptive and comprehensive results. Moreover, interpreting results is difficult if one is unaware of the significant properties of the benchmarks at hand, especially if exceptional or unexpected behavior must be explained. Consequently, researchers tediously reexamine benchmarks to find properties that might explain observed behavior.

The goal and contribution of this work is to provide a description of commonly used benchmarks in terms of memory behavior and garbage collection behavior. We do not want to encourage cherry-picking benchmarks but rather enable researchers and reviewers to better evaluate the work of others. We will show important properties of popular Java benchmarks, as well as curiosities one should be aware of when using them and when evaluating other work based on these benchmarks. Furthermore, we split the benchmarks into categories, depending on observed properties, such as allocated memory, survivor ratios, live sizes, and garbage collection times under different virtual machine configurations. We selected DaCapo, DaCapo Scala and SPECjvm2008 because they are the most popular Java benchmark suites, they are free to use, and they are open source, which makes them ideal for scientific evaluations.

Since the observer effect in state-of-the-art monitoring approaches distorts the application behavior significantly (cf. Bitto et al. [1] for detailed analysis on the observer effect in memory and GC monitoring), we will base our analysis on AntTracks, a memory monitoring tool that is accurate at the

[1]http://www.dacapobench.org/
[2]http://www.dacapo.scalabench.org/
[3]https://www.spec.org/jvm2008/
[4]https://www.spec.org/jbb2015/

object-level and imposing only very low run-time overhead, first introduced by Lengauer et al. [6].

This paper is structured as follows: In Section 2 we describe our research methodology. Section 3 shows different metrics with respect to an application's allocation behavior and garbage collection behavior per benchmark suite. Section 4 presents related work, while Section 5 concludes this paper.

## 2. METHODOLOGY

This section describes the research methodology, i.e., the benchmarks we used and their configuration, the hardware setup, as well as the method of measurement.

### *Benchmarks.*

Figure 1 shows all benchmarks of the DaCapo (version 9.12), DaCapo Scala (version 0.1.0-20120216.103539-3) and SPECjvm2008 benchmark suite, i.e., all benchmarks we will examine in this paper. It also shows the input used, the number of warmups performed before measurement, and the live size of every benchmark. The live size is defined as the maximum number of bytes alive at any given point in time throughout the execution of the benchmark. The input size is either identified by name (for DaCapo and DaCapo Scala) or by the number of operations (for SPECjvm2008, based on the concurrent lagom workload). The number of warmups, i.e., the number of times we executed the benchmark before measuring results, is ideally 20, however, we increased or decreased the warmups of specific benchmarks based on the input. We chose 20 (modified according to the input size) as the baseline for the warmups because the DaCapo suite's built-in mechanism automatically detects a steady state after at most this amount of warmups. Furthermore, we needed to increase the number of warmups until JIT-compilation and GC ergonomics also stabilized. As most benchmarks completely stabilized (i.e., DaCapo built-in convergence, JIT-compilation, GC ergonomics, and heap space resizing) after a similar number of warmups (close to 20, or a factor thereof depending on the input size), we chose 20 as a round number of wamups for all benchmarks. This number also prooved useful for the SPECjvm benchmarks built-in warmup mechanism runs for the same amount of time (the SPECjvm warmup mechanism is based on time rather than iterations). Some DaCapo and DaCapo Scala benchmarks have *huge* and *gargantuan* workloads, which we used to put more pressure on the memory system. In these cases we were able to decrease the number of warmups. Other benchmarks do not have large loads, or the large and default loads crash on Java 8 (because they rely on internal classes that do not exist any longer). In these cases we used the biggest functioning workload, and increased the number of warmups accordingly. The live size has been determined by finding the lowest maximum heap setting with which the benchmarks will execute without an `OutOfMemoryError`. This live size was determined by trial and error, i.e., binary searching the lowest possible heap setting with the `Xmx` flag with 1KB granularity. We will use a multiple of the live size to determine a realistic heap limit.

| | Benchmark | Warmups | Input | Live[MB] |
|---|---|---|---|---|
| DaCapo | avrora | 20 | large | 7.49 |
| | batik | 80 | small | 24.74 |
| | eclipse | 80 | small | 14.26 |
| | fop | 40 | default | 29.57 |
| | h2 | 10 | huge | 1300.67 |
| | jython | 20 | large | 27.93 |
| | luindex | 40 | default | 5.03 |
| | lusearch | 20 | large | 2.64 |
| | pmd | 20 | large | 38.06 |
| | sunflow | 20 | large | 11.06 |
| | tomcat | 10 | huge | 17.02 |
| | tradebeans | 10 | huge | 278.42 |
| | tradesoap | 10 | huge | 110.81 |
| | xalan | 20 | large | 5.16 |
| DaCapo Scala | actors | 5 | gargantuan | 17.02 |
| | apparat | 5 | gargantuan | 66.68 |
| | factorie | 5 | gargantuan | 558.27 |
| | kiama | 40 | default | 6.45 |
| | scalac | 20 | large | 71.86 |
| | scaladoc | 20 | large | 68.44 |
| | scalap | 20 | large | 5.76 |
| | scalariform | 10 | huge | 19.15 |
| | scalaxb | 10 | huge | 109.06 |
| | tmt | 10 | huge | 39.22 |
| SPECjvm2008 | compiler.compiler | 20 | 160ops | 229.10 |
| | compiler.sunflow | 20 | 160ops | 144.32 |
| | compress | 20 | 40ops | 85.20 |
| | crypto.aes | 20 | 16ops | 41.75 |
| | crypto.rsa | 20 | 120ops | 3.19 |
| | crypto.signverify | 20 | 96ops | 18.32 |
| | derby | 20 | 240ops | 435.10 |
| | mpegaudio | 20 | 40ops | 4.49 |
| | scimark.fft.large | 20 | 8ops | 612.17 |
| | scimark.fft.small | 20 | 80ops | 16.46 |
| | scimark.lu.large | 20 | 8ops | 583.21 |
| | scimark.lu.small | 20 | 96ops | 10.78 |
| | scimark.monte_carlo | 20 | 72ops | 2.93 |
| | scimark.sor.large | 20 | 8ops | 294.84 |
| | scimark.sor.small | 20 | 65ops | 6.70 |
| | scimark.sparse.large | 20 | 8ops | 446.93 |
| | scimark.sparse.small | 20 | 16ops | 11.45 |
| | serial | 20 | 200ops | 367.17 |
| | sunflow | 20 | 120ops | 19.26 |
| | xml.transform | 20 | 56ops | 35.12 |
| | xml.validation | 20 | 320ops | 92.79 |

**Figure 1: Benchmarks and their respective warmups, inputs (the name of the input for DaCapo and DaCapo Scala, and the number of operations for SPECjvm2008), and live sizes.**

### *Setup.*

All measurements were run on an Intel® Core(TM) i7-4770 CPU @ 3.40GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 840 PRO Series (DXM03B0Q), running Ubuntu Wily Werewolf 15.10 with the Kernel Linux 4.2.0-25-generic. All unnecessary services were disabled in order not to distort the experiments.

### *Measurement.*

All numbers reported in this paper represent the steady-state performance of 50 runs, based on Georges et al. [4, 5]. Every benchmark has been warmed up before measurement

to stabilize caches as well as JIT optimization. Consequently, we report peak performance only, i.e., we exclude all warmup runs and present the best value of the 50 measurements, whereupon the best run is defined as the measurment with the minimal run time. Also, we forced a garbage collection before every measurement to collect leftovers from previous warmups. This collection is not included in the measurement (because it would not occur naturally would we not force it).

In order to record global performance numbers, such as run time, compilation time, as well as garbage collection frequency and time, we used a custom agent hooking into the VM using the Java Virtual Machine Tool Interface (JVMTI) in combination with benchmark-specific mechanisms (DaCapo Callbacks and SPECjvm2008 Analyzers). We were very careful not to enable any capabilities in the VM that would change the VM's behavior.

To make more detailed measurements, we used AntTracks (version 20160101), a special HotSpot-based VM (based on OpenJDK 8u102), that records a trace of memory events almost without changing the VM's behavior. Based on the generated trace, we extracted metrics such as allocated objects, survivor ratios and top allocation sites.

AntTracks is able to record memory traces almost without changing the original behavior because it is implemented, compared to other state-of-the-art tools, directly into the VM and uses a very efficient trace format. It does not have to deal with heavy-weight instrumentation that impedes escape analysis and it also does not have to introduce `WeakReference`s for finalizers to detect object deallocations. The VM and especially the GC's are modified to efficiently generate a very compact event trace, omitting everything that can be reconstructed offline. A dedicated offline tool then postprocesses the trace and reconstructs everything that has been omitted. Moreover, the trace is complete, i.e., it does not miss a single object allocation of internal GC operation. Thus, the only disruption that might occur is a small overall overhead around 4%.

# 3. STUDY

The following sections describe the memory and garbage collection behavior of the benchmarks with a heap that is limited to three times the respective live size (adaptive heap limit). For evaluating the garbage collection behavior we used the *ParallelOld GC*, the current default collector in the HotSpot™ VM. Figures for the concurrent *G1 GC*, the designated default collector in Java SE 9 allowing the handling of big heaps more efficiently, can be found in the appendix.

## 3.1 Allocation Behavior

This section examines the allocation behavior of every benchmark in detail. As allocations are only marginally influenced by the GC, we report only the results with an adaptive heap and the ParallelOld GC.

Figure 2 shows the basic allocation behavior of all examined benchmarks. It presents the number of allocations of every benchmark (final measurement iteration only, VM startup and warmups excluded), in number of objects as well as in bytes. The table also shows the number of objects and the amount of memory allocated per second, as well as the average object layout, i.e., the average object size, the percentage of array objects and their average length.

### *Total Allocations.*

Measuring the total number of allocations (per iteration) shows us that factorie, serial, tmt, sunflow (SPECjvm2008) and derby are the 5 most allocation-intensive benchmarks. They allocate up to $5.7 * 10^9$ objects in a single iteration, which puts significant pressure on the GC. Looking at the amount of memory being allocated, the 5 most memory-intensive benchmarks are again factorie, serial, sunflow, derby, and tmt. Please note, that factorie and sunflow (SPECjvm2008) allocate up to 137GB and 134GB of memory respectively.

This metric is of particular interest when developing new allocation algorithms or instrumentation-based memory monitoring tools. Memory monitoring tools often instrument every `new` instruction to record allocations. They collect information such as the type, size, or length in case of an array. As these tools want to distort the application's behavior as little as possible in order to present the user with accurate analyses, benchmarks exhibiting a high number of allocations are interesting to use as a baseline for overhead measurements.

### *Allocation Rate.*

Measuring object allocations per second (i.e., the number of objects allocated within one iteration divided by the run time of that iteration), the 5 most intensive benchmarks are derby, serial, tmt, factorie, and xml.transform (please note, that there are several other benchmarks that are only slightly below) with up to $3 * 10^7$ new objects per second. This number will have a direct impact on the garbage collection frequency (cf. Section 3.3). Looking at the amount of memory allocated every second, the most intensive benchmarks are derby, serial, sunflow (SPECjvm2008), tmt, and xml.transform.

Similarly to the total number of allocations, this metric is of interest for VM implementors and monitoring tool developers alike. Moreover, the allocation rate will have a significant impact if the underlying hardware is not fast enough, i.e., memory monitoring tools may write their data to disk. In addition, this metric is of interest for GC developers because a high allocation rate will lead to a high GC frequency. Also, the allocation rate may be a vital basic information for manually tuning GC parameters, i.e., to minimize the overall GC time or to keep the maximum pause time below a predefined threshold.

### *Object Layout.*

In contrast to the total number of allocations, the largest objects are allocated by the scimark.fft.large benchmark. However, since all scimark benchmarks have a short run time and a small allocation rate in common, their actual pressure on the memory is low. The ratios of instances and arrays vary widely, ranging from 3.2 % arrays in the case of sunflow (SPECjvm2008) to up to 97.5 % in the case of mpegaudio.

Compared to the DaCapo benchmark suite and the DaCapo Scala benchmark suite, the SPECjvm2008 benchmarks show a much higher average array length in general. This is due to the fact, that many benchmarks work on either big input data (e.g., compress, crypto.*, and mpegaudio) or on big matrices (e.g., scimark.*), and, consequently, create many big arrays.

| | Benchmark | Allocations | | | | Object layout | | |
|---|---|---|---|---|---|---|---|---|
| | | [$10^3$] | [MB] | [$10^3$ / sec] | [MB / sec] | Avg obj size [b] | Array rate [%] | Avg array length |
| DaCapo | avrora | 6,710.1 | 204.4 | 828.5 | 25.2 | 30.5 | 31.8 | 12.1 |
| | batik | 462.3 | 32.4 | 2,454.0 | 171.9 | 70.0 | 38.6 | 112.3 |
| | eclipse | 676.6 | 0.0 | 6,346.7 | 0.0 | 61.8 | 49.1 | 120.6 |
| | fop | 2,511.8 | 106.8 | 14,435.4 | 613.9 | 42.5 | 35.1 | 19.8 |
| | h2 | 388,081.9 | 14,838.3 | 2,498.6 | 95.5 | 38.2 | 38.5 | 10.8 |
| | jython | 180,707.2 | 7,837.1 | 18,653.8 | 809.0 | 43.4 | 32.1 | 58.4 |
| | luindex | 217.4 | 10.6 | 618.4 | 30.3 | 48.9 | 38.0 | 182.7 |
| | lusearch | 21,904.6 | 2,270.3 | 4,565.7 | 473.2 | 103.6 | 44.0 | 553.0 |
| | pmd | 15,660.6 | 565.4 | 11,231.9 | 405.5 | 36.1 | 40.3 | 25.4 |
| | sunflow | 138,674.2 | 6,254.3 | 24,670.7 | 1,112.7 | 45.1 | 3.4 | 10.5 |
| | tomcat | 177,454.6 | 9,509.6 | 5,724.9 | 306.8 | 53.6 | 48.1 | 106.0 |
| | tradebeans | 925,570.5 | 38,817.5 | 15,678.1 | 657.5 | 40.0 | 40.0 | 14.3 |
| | tradesoap | 944,162.7 | 45,057.8 | 15,178.5 | 724.4 | 47.7 | 36.5 | 92.2 |
| | xalan | 103,069.1 | 4,989.3 | 5,614.9 | 271.8 | 48.4 | 40.3 | 60.9 |
| DaCapo Scala | actors | 245,235.3 | 6,002.2 | 15,843.8 | 387.8 | 24.5 | 3.9 | 6.8 |
| | apparat | 399,087.3 | 12,837.0 | 4,433.0 | 142.6 | 32.2 | 24.9 | 16.3 |
| | factorie | 5,716,589.7 | 137,521.8 | 37,494.4 | 902.0 | 24.1 | 5.4 | 3.9 |
| | kiama | 11,384.6 | 0.0 | 25,885.9 | 0.0 | 35.0 | 27.4 | 22.5 |
| | scalac | 42,252.8 | 1,335.0 | 13,752.4 | 434.5 | 31.6 | 18.9 | 21.5 |
| | scaladoc | 38,065.2 | 1,471.3 | 14,958.0 | 578.2 | 38.7 | 30.6 | 34.8 |
| | scalap | 3,454.1 | 87.4 | 16,201.3 | 409.7 | 25.3 | 12.3 | 39.2 |
| | scalariform | 50,745.1 | 1,259.1 | 21,352.9 | 529.8 | 24.8 | 17.6 | 7.3 |
| | scalaxb | 99,651.5 | 2,464.6 | 8,419.4 | 208.2 | 24.7 | 4.6 | 107.8 |
| | tmt | 2,663,579.5 | 65,294.7 | 54,697.7 | 1,340.9 | 24.5 | 0.8 | 62.4 |
| SPECjvm2008 | compiler.compiler | 471,781.5 | 15,712.4 | 19,408.0 | 646.4 | 33.3 | 13.0 | 41.7 |
| | compiler.sunflow | 1,195,208.0 | 42,073.5 | 17,498.7 | 616.0 | 35.2 | 16.9 | 42.3 |
| | compress | 36.7 | 64.9 | 6.3 | 11.2 | 1,767.8 | 40.8 | 15,220.5 |
| | crypto.aes | 73.9 | 368.2 | 8.8 | 43.8 | 4,979.5 | 63.6 | 61,137.9 |
| | crypto.rsa | 32,880.1 | 2,081.6 | 2,583.7 | 163.6 | 63.3 | 59.6 | 26.7 |
| | crypto.signverify | 4,009.0 | 867.1 | 383.3 | 82.9 | 216.3 | 65.9 | 1,598.2 |
| | derby | 2,001,219.7 | 77,974.7 | 86,931.7 | 3,387.2 | 39.0 | 20.6 | 14.9 |
| | mpegaudio | 561.1 | 279.8 | 54.2 | 27.0 | 498.7 | 97.5 | 3,916.1 |
| | scimark.fft.large | 1.1 | 72.6 | 0.1 | 9.2 | 66,335.2 | 45.6 | 157,485.6 |
| | scimark.fft.small | 87.1 | 355.5 | 11.1 | 45.4 | 4,080.7 | 66.9 | 6,382.7 |
| | scimark.lu.large | 34.5 | 68.5 | 1.0 | 1.9 | 1,985.1 | 98.3 | 2,060.3 |
| | scimark.lu.small | 4,023.3 | 875.1 | 309.2 | 67.3 | 217.5 | 99.1 | 222.6 |
| | scimark.monte_carlo | 13.4 | 1.5 | 1.4 | 0.2 | 108.1 | 41.3 | 748.4 |
| | scimark.sor.large | 17.8 | 34.2 | 2.5 | 4.9 | 1,916.6 | 96.9 | 2,021.2 |
| | scimark.sor.small | 11.4 | 1.6 | 1.7 | 0.2 | 141.5 | 70.1 | 477.0 |
| | scimark.sparse.large | 1.1 | 53.9 | 0.1 | 4.8 | 50,820.6 | 48.6 | 141,717.1 |
| | scimark.sparse.small | 6.2 | 38.1 | 1.9 | 11.4 | 6,154.3 | 66.0 | 12,441.3 |
| | serial | 3,361,406.2 | 134,122.4 | 64,576.1 | 2,576.6 | 39.9 | 48.6 | 16.2 |
| | sunflow | 2,047,771.6 | 93,002.4 | 29,547.1 | 1,341.9 | 45.4 | 3.2 | 9.7 |
| | xml.transform | 253,054.8 | 9,716.2 | 30,966.1 | 1,189.0 | 38.4 | 39.1 | 38.6 |
| | xml.validation | 807,942.9 | 27,876.6 | 30,325.7 | 1,046.3 | 34.5 | 33.5 | 12.9 |

**Figure 2: Allocations total and per second as well as the average object layout (size, array rate, array length)**

## 3.2 Allocating Subsystems

Figure 3 shows the percentage of objects allocated by VM-internal code (e.g., native code or filler objects for keeping the heap unfragmented), of objects allocated by interpreted code (i.e., code that has not been deemed worth compiling yet), of objects allocated by C1-compiled code (i.e., code that has been compiled by the client compiler), and of objects allocated by C2-compiled code (i.e., code that has been compiled by the server compiler). It also shows the overall compile time ratio, i.e., the time the application spent on compiling in relation to the overall run time.

### *Allocating Code.*

When executing a method for the first time, the VM starts interpreting the code without applying any optimizations. During interpretation, statistics about the method are recorded, e.g., execution counters, branch frequencies, and value ranges for variables. Using these statistics, the method will eventually be compiled by the client compiler. This compiler applies some optimizations and also inserts code to continue recording statistics about executions. Finally, if the method is used often enough, it will be compiled by the server compiler. The server compiler will apply more aggressive optimizations and makes assumptions based on observations made by the interpreter or by the client-compiled code. Should an assumption turn out to be wrong, the compiled code will be discarded and the VM will fall back to the interpreter for this method and start over recording new statistics. Eventually, the VM will retry compilation.

Looking at how much of the allocating code is compiled can tell us how well the benchmark is warmed up because we do not want the VM to execute unoptimized methods or to spend time on compiling while we are measuring. Our measurements show that all benchmarks except some of the SPECjvm2008 scimark.* benchmarks have been properly warmed up with our configuration.

The scimark benchmarks perform mathematical computations that are not very allocation intensive. They consist of small amounts of code operating only on primitive matrices as data structures if any. Consequently, they have an unusually big memory-to-object ratio. Moreover, as they have many long-running methods doing number crunching in loops, it takes some time until they reach an execution frequency that is high enough for triggering compilation. However, because they have neither a lot of total allocations nor a large allocation rate, they are not very useful for examining allocation behavior anyway.

The lusearch benchmark reports 13.7% VM-internal allocations due to the unlucky use of Exceptions in normal control flow. Creating an object of type `Throwable` (superclass of all exceptions and errors) results in a call to the native method `fillInStackTrace`. This method walks the stack, creates several `Object`, `short`, and `int` arrays containing the objects, methods, and the corresponding bci offsets on the stack, and finally puts those arrays in the `Throwable` object. The lusearch benchmark allocates more than 99.9% of all VM-internally allocated objects by filling the stack trace of an unnecessary exception, the rest are mostly application domain objects created by cloning. Figure 4 shows the stack of an exception used to indicate the end of a character-based stream.

Like lusearch, the pmd benchmark allocates about 90% of all VM-internal objects by filling the stack trace of an

| | Benchmark | Allocated by [%] VM | Int. | C1 | C2 | Comp. [%] |
|---|---|---|---|---|---|---|
| DaCapo | avrora | 1.1 | 0.0 | 0.5 | 98.3 | 0.11 |
| | batik | 0.6 | 0.1 | 1.8 | 97.5 | 1.06 |
| | eclipse | 4.6 | 0.0 | 2.4 | 93.0 | 0.00 |
| | fop | 0.2 | 0.1 | 0.9 | 98.9 | 24.48 |
| | h2 | 1.0 | 0.0 | 0.0 | 99.0 | 3.64 |
| | jython | 2.1 | 0.0 | 0.0 | 97.9 | 0.85 |
| | luindex | 1.6 | 0.3 | 0.8 | 97.3 | 0.70 |
| | lusearch | 13.7 | 0.0 | 0.1 | 86.2 | 0.12 |
| | pmd | 11.6 | 0.0 | 0.1 | 88.3 | 2.91 |
| | sunflow | 0.2 | 0.0 | 0.0 | 99.8 | 0.01 |
| | tomcat | 1.8 | 0.0 | 0.5 | 97.7 | 6.80 |
| | tradebeans | 0.5 | 0.0 | 0.0 | 99.5 | 3.33 |
| | tradesoap | 0.4 | 0.0 | 0.0 | 99.6 | 0.25 |
| | xalan | 1.8 | 0.0 | 0.0 | 98.2 | 0.01 |
| DaCapo Scala | actors | 0.2 | 0.0 | 0.0 | 99.7 | 0.39 |
| | apparat | 0.1 | 0.0 | 0.1 | 99.8 | 0.37 |
| | factorie | 0.0 | 0.0 | 0.0 | 100.0 | 0.18 |
| | kiama | 0.0 | 0.0 | 0.3 | 99.7 | 9.37 |
| | scalac | 0.1 | 0.3 | 0.6 | 99.0 | 23.59 |
| | scaladoc | 0.2 | 0.0 | 0.7 | 99.1 | 13.09 |
| | scalap | 0.1 | 0.0 | 0.6 | 99.3 | 10.14 |
| | scalariform | 0.0 | 0.0 | 0.5 | 99.5 | 4.72 |
| | scalaxb | 0.0 | 0.0 | 0.1 | 99.9 | 1.54 |
| | tmt | 0.0 | 0.0 | 0.0 | 100.0 | 0.02 |
| SPECjvm2008 | compiler.compiler | 0.1 | 0.0 | 0.0 | 99.9 | 0.11 |
| | compiler.sunflow | 0.2 | 0.0 | 0.0 | 99.8 | 0.00 |
| | compress | 1.4 | 0.3 | 29.4 | 68.9 | 0.05 |
| | crypto.aes | 4.3 | 0.3 | 8.9 | 86.5 | 0.08 |
| | crypto.rsa | 2.6 | 0.0 | 0.0 | 97.4 | 0.07 |
| | crypto.signverify | 1.9 | 0.0 | 0.1 | 98.0 | 0.01 |
| | derby | 0.0 | 0.0 | 0.0 | 100.0 | 0.06 |
| | mpegaudio | 5.1 | 0.0 | 1.3 | 93.6 | 0.12 |
| | scimark.fft.large | 4.9 | 39.1 | 42.3 | 13.7 | 0.05 |
| | scimark.fft.small | 5.8 | 0.2 | 4.2 | 89.8 | 0.11 |
| | scimark.lu.large | 2.0 | 48.7 | 1.4 | 48.0 | 0.03 |
| | scimark.lu.small | 20.3 | 0.0 | 0.1 | 79.6 | 0.05 |
| | scimark.monte_carlo | 1.6 | 1.0 | 34.7 | 62.7 | 0.09 |
| | scimark.sor.large | 2.7 | 94.3 | 2.1 | 0.8 | 0.42 |
| | scimark.sor.small | 8.1 | 18.9 | 30.8 | 42.2 | 0.01 |
| | scimark.sparse.large | 9.0 | 38.5 | 31.3 | 21.2 | 0.02 |
| | scimark.sparse.small | 20.3 | 5.2 | 21.6 | 53.0 | 7.23 |
| | serial | 0.0 | 0.0 | 0.0 | 100.0 | 0.00 |
| | sunflow | 0.2 | 0.0 | 0.0 | 99.8 | 0.02 |
| | xml.transform | 0.1 | 0.0 | 0.0 | 99.9 | 0.36 |
| | xml.validation | 0.6 | 0.0 | 0.0 | 99.4 | 0.02 |

**Figure 3: Objects allocated by VM-internal code, interpreted code, C1 compiled code, or by C2 compiled code respectively (green: 1st top allocator, yellow: 2nd top allocator, red: 3rd top allocator), as well as the time spent compiling in relation to the overall run time**

```
java.lang.Throwable.fillInStackTrace():16
java.lang.Throwable.<init>():24
java.lang.Exception.<init>():2
java.io.IOException.<init>():2
org.apache.lucene.queryParser.
    FastCharStream.refill():156
org.apache.lucene.queryParser.
    FastCharStream.readChar():12
org.apache.lucene.queryParser.
    FastCharStream.BeginToken():9
org.apache.lucene.queryParser.
    QueryParserTokenManager.getNextToken()
    :7
...
```

**Figure 4: Stack of an exception used to steer control flow in the lusearch benchmark, resulting in about 2,629,000 objects for representing the stack traces**

exception. The other 10% of VM-internal allocations are `Strings` and the corresponding `char[]` that are used to look up classes using an `URLClassLoader`. Figures 5 and 6 show the stack of an exception when using the `URLClassLoader`.

```
java.lang.Throwable.fillInStackTrace():16
java.lang.Throwable.<init>():24
java.lang.Exception.<init>():3
java.lang.ReflectiveOperationException.<
    init>():3
java.lang.ClassNotFoundException.<init>()
    :3
java.net.URLClassLoader.findClass():41
java.lang.ClassLoader.loadClass():70
org.dacapo.harness.DacapoClassLoader.
    loadClass():24
java.lang.ClassLoader.loadClass():38
java.lang.ClassLoader.loadClass():3
net.sourceforge.pmd.typeresolution.
    ClassTypeResolver.processOnDemand():56
```

**Figure 5: Stack of an exception used to steer control flow in the pmd benchmark, resulting in 1,598,438 objects for representing the stack traces**

```
java.lang.ClassLoader.findLoadedClass():12
java.lang.ClassLoader.loadClass():10
sun.misc.Launcher$AppClassLoader.loadClass
    ():81
java.lang.ClassLoader.loadClass():38
org.dacapo.harness.DacapoClassLoader.
    loadClass():24
java.lang.ClassLoader.loadClass():3
net.sourceforge.pmd.symboltable.
    ScopeAndDeclarationFinder.
    createClassScope():23
```

**Figure 6: Stack of class loading by the `URLClassLoader`, resulting in 193,682 objects for looking up a class by name**

*Compile time.*

The JIT compile time ratio (i.e., the summed up compile time of all compilation threads within an iteration divided by the cpu time of that iteration) shows that, if the application has been warmed up properly, the compile time is negligible in most benchmarks. Some benchmarks, however, i.e., fop, scalac, and scalap, show a high compilation time although almost all allocations are already executed by compiled code. This indicates, that these benchmarks should probably be warmed up better if run time performance is measured. However, in these cases we stuck to the selected methodology to be still comparable.

## 3.3 Garbage Collection

This section examines the garbage collection behavior of every benchmark in detail. For comparability we provide figures for a heap limited to three times the benchmark's live size (see Figure 7). For a fixed heap size of 1GB and for a heap that is unlimited, please refer to Figures 9 and 10 in the Appendix.

Figure 7 shows the garbage collection count, the total garbage collection time, and the average pause time for the ParallelOld GC and the G1 GC. All metrics are split into values for *minor* (-) and *major* (+) collections. In a major collection, the entire heap is collected, whereas a minor collection collects only parts of the heap (the young generation in case of the ParallelOld GC, any subset of regions in case of the G1 GC).

*GC Count.*

The benchmarks differ widely in terms of GC count, from scimark.fft.large without any collection, to lusearch with 7041 collections. Only some benchmarks perform major collections with the ParallelOld GC.

In general, there is an easy-to-see 97% correlation (linear Pearson correlation) between the number of collections (minor and major) of the ParallelOld GC and the G1 GC. However, the G1 GC usually performs less collections than the ParallelOld GC with only a few exceptions, i.e., compiler.sunflow, crypto.rsa, scimark.fft.small, scimark.lu.*, scimark.sor.large, and serial. This is due to the fact, that G1 can select which heap regions to collect. Consequently, G1 selects regions with a lot of garbage, resulting in more memory being freed. Moreover, G1 can include regions of the old generation in a minor collection, whereas the ParallelOld GC can collect the old generation only with a major collection. This behavior can reduce floating garbage (old dead objects keeping young objects alive) significantly.

In contrast to the ParallelOld GC, where a major collection normally occurs after some minor collections, the G1 GC uses major collection only as an emergency action. For this reason, G1 major collections are so rare and occur only in 4 benchmarks, i.e., lusearch, xalan, scimark.fft.small, and scimark.lu.small. These benchmarks have a very small live set, and consequently a very low heap limit. This shows that G1, although performing well in most cases, is not built to handle small heaps efficiently.

For evaluating garbage collection algorithms which depend on the number of live objects, only allocation-intensive benchmarks with short-living objects are advisable, e.g., tmt and serial. Allocation-intense benchmarks with long-living objects, e.g., derby, are recommendable to test the performance of compaction algorithms, e.g., Mark & Compact.

This metric is interesting to test monitoring tools and GCs which introduce overhead per collection. Especially considering stop-the-world GCs may introduce a significant

| | Benchmark | ParallelOld GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] | G1 GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DaCapo | avrora | 68 | 0 | 1.5 | - | 1 | - | 50 | 48 | 0 | 0.7 | - | 1 | - | 76 |
| | batik | 10 | 0 | 11.7 | - | 2 | - | 49 | 3 | 0 | 6.5 | - | 3 | - | 89 |
| | eclipse | 1 | 0 | 1.8 | - | 2 | - | 531 | 2 | 0 | 3.4 | - | 2 | - | 376 |
| | fop | 10 | 0 | 24.1 | - | 4 | - | 58 | 5 | 0 | 12.1 | - | 3 | - | 112 |
| | h2 | 28 | 0 | 5.4 | - | 300 | - | 94 | 28 | 0 | 1.4 | - | 81 | - | 129 |
| | jython | 468 | 0 | 11.3 | - | 2 | - | 111 | 255 | 0 | 5.8 | - | 2 | - | 231 |
| | luindex | 11 | 0 | 7.5 | - | 2 | - | 37 | 5 | 0 | 2.3 | - | 1 | - | 91 |
| | lusearch | 6,979 | 62 | 56.0 | 12.7 | 0 | 9 | 36 | 4,748 | 1,431 | 23.9 | 64.2 | 0 | 7 | 82 |
| | pmd | 50 | 0 | 17.8 | - | 4 | - | 66 | 20 | 0 | 5.5 | - | 3 | - | 137 |
| | sunflow | 771 | 8 | 15.9 | 1.6 | 1 | 11 | 53 | 559 | 0 | 12.5 | - | 1 | - | 129 |
| | tomcat | 1,234 | 0 | 4.4 | - | 1 | - | 114 | 680 | 0 | 1.9 | - | 0 | - | 293 |
| | tradebeans | 178 | 1 | 14.2 | 1.0 | 47 | 590 | 49 | 105 | 0 | 5.0 | - | 29 | - | 103 |
| | tradesoap | 961 | 12 | 46.0 | 4.9 | 29 | 254 | 50 | 981 | 0 | 25.4 | - | 14 | - | 93 |
| | xalan | 4,607 | 422 | 45.8 | 30.5 | 1 | 13 | 31 | 3,369 | 337 | 32.7 | 29.8 | 1 | 13 | 50 |
| DaCapo Scala | actors | 388 | 1 | 1.9 | 0.0 | 0 | 14 | 53 | 264 | 0 | 1.2 | - | 0 | - | 143 |
| | apparat | 259 | 0 | 1.7 | - | 6 | - | 73 | 148 | 0 | 0.8 | - | 4 | - | 148 |
| | factorie | 272 | 0 | 41.1 | - | 253 | - | 180 | 171 | 0 | 1.4 | - | 9 | - | 100 |
| | kiama | 28 | 0 | 43.9 | - | 6 | - | 59 | 23 | 0 | 22.0 | - | 3 | - | 69 |
| | scalac | 46 | 0 | 25.2 | - | 16 | - | 112 | 22 | 0 | 7.4 | - | 9 | - | 146 |
| | scaladoc | 53 | 0 | 26.8 | - | 12 | - | 138 | 26 | 0 | 7.6 | - | 6 | - | 149 |
| | scalap | 40 | 0 | 31.6 | - | 1 | - | 90 | 13 | 0 | 12.7 | - | 1 | - | 276 |
| | scalariform | 127 | 0 | 21.0 | - | 3 | - | 124 | 48 | 0 | 6.8 | - | 3 | - | 204 |
| | scalaxb | 50 | 0 | 2.2 | - | 5 | - | 176 | 35 | 0 | 1.2 | - | 4 | - | 160 |
| | tmt | 2,326 | 0 | 4.5 | - | 0 | - | 100 | 1,597 | 0 | 3.7 | - | 1 | - | 123 |
| SPECjvm2008 | compiler.compiler | 155 | 11 | 50.2 | 10.9 | 78 | 243 | 41 | 119 | 0 | 32.7 | - | 54 | - | 95 |
| | compiler.sunflow | 612 | 54 | 49.4 | 13.9 | 55 | 175 | 43 | 638 | 0 | 40.9 | - | 35 | - | 68 |
| | compress | 14 | 0 | 0.1 | - | 0 | - | 49 | 4 | 0 | 0.0 | - | 1 | - | 177 |
| | crypto.aes | 163 | 2 | 2.8 | 0.2 | 1 | 8 | 49 | 64 | 0 | 2.6 | - | 3 | - | 6 |
| | crypto.rsa | 1,047 | 2 | 3.6 | 0.1 | 0 | 7 | 35 | 1,078 | 0 | 3.7 | - | 0 | - | 171 |
| | crypto.signverify | 715 | 32 | 7.0 | 2.2 | 1 | 7 | 39 | 407 | 0 | 5.0 | - | 1 | - | 21 |
| | derby | 181 | 0 | 3.5 | - | 4 | - | 95 | 103 | 0 | 1.4 | - | 3 | - | 184 |
| | mpegaudio | 683 | 7 | 2.8 | 0.4 | 0 | 6 | 46 | 458 | 0 | 2.2 | - | 0 | - | 109 |
| | scimark.fft.large | 0 | 0 | - | - | - | - | - | 1 | 0 | 0.0 | - | 1 | - | 0 |
| | scimark.fft.small | 322 | 0 | 2.5 | - | 0 | - | 108 | 381 | 1 | 4.4 | 0.0 | 0 | 6 | 15 |
| | scimark.lu.large | 2 | 0 | 0.1 | - | 33 | - | 191 | 13 | 0 | 0.2 | - | 7 | - | 54 |
| | scimark.lu.small | 1,738 | 377 | 14.5 | 20.1 | 1 | 6 | 34 | 2,353 | 44 | 16.4 | 2.6 | 0 | 8 | 45 |
| | scimark.monte_carlo | 3 | 0 | 0.0 | - | 0 | - | 115 | 1 | 0 | 0.0 | - | 1 | - | 261 |
| | scimark.sor.large | 2 | 0 | 0.4 | - | 17 | - | 190 | 12 | 0 | 0.7 | - | 4 | - | 39 |
| | scimark.sor.small | 3 | 0 | 0.0 | - | 0 | - | 74 | 4 | 0 | 0.0 | - | 1 | - | 77 |
| | scimark.sparse.large | 3 | 0 | 0.5 | - | 20 | - | 127 | 1 | 0 | 0.0 | - | 1 | - | 0 |
| | scimark.sparse.small | 70 | 2 | 1.4 | 0.3 | 0 | 6 | 34 | 45 | 0 | 1.4 | - | 1 | - | 74 |
| | serial | 384 | 0 | 0.9 | - | 1 | - | 128 | 1,156 | 0 | 1.6 | - | 0 | - | 47 |
| | sunflow | 5,500 | 40 | 10.2 | 1.0 | 1 | 18 | 51 | 3,929 | 0 | 10.2 | - | 1 | - | 132 |
| | xml.transform | 562 | 9 | 13.1 | 3.0 | 1 | 27 | 50 | 333 | 0 | 7.8 | - | 1 | - | 126 |
| | xml.validation | 608 | 27 | 23.2 | 13.8 | 10 | 136 | 51 | 442 | 0 | 16.3 | - | 9 | - | 68 |

**Figure 7: GC count, GC time relative to the total run time (green: less than 5%, yellow: less than 15%, red: more than 15%), and average pause time for minor (-) and major (+) GCs for the ParallelOld GC and the G1 GC respectively (green: less than 10ms, yellow: less than 100ms, red: more than 100ms)**

overhead when all application threads need to be suspended repeatedly. Also, this metric can be used to tune the GC behavior.

*GC Time.*

The GC time, i.e., the percentage the application spends on garbage collection relative to its entire run time, depends on the number of collections as well as on the length of every collection. For G1, which marks concurrently, this metric only includes the time the application was paused (the G1 GC is not fully concurrent but pauses the application to move objects). The measurements show that a benchmark can easily spend more than 50% of its total run time on garbage collection. This illustrates that, even if memory performance is not one's primary concern, it must be dealt with.

Again, the G1 GC performs better in most cases with the same exceptions as for the GC count. Especially, in some cases with a high GC time, the G1 performs a lot better (e.g., factorie 41% vs 1.4%).

Obviously, this metric is of interest for GC developers for optimizing garbage collection algorithms. Also, memory monitoring tools that rely on detecting deallocations with the help of `WeakReference`s or finalizers can make good use of benchmarks with a high GC time. As `WeakReference`s and finalizers introduce a lot of additional work for the GC, benchmarks with an already high GC time will be interesting for overhead measurements.

*GC Pauses.*

The length of a GC pause depends on the complexity of the underlying collection algorithm. The ParallelOld GC and the G1 GC use different algorithms for minor and major collections respectively.

The complexity of a minor collection (in the ParallelOld GC as well as in the G1 GC) mostly depends on the number of live objects residing in the collected regions, as they must be evacuated to the survivor space or promoted to the old generation. The complexity of a major collection (in the ParallelOld GC as well as in the G1 GC) depends on the total number of objects, as both algorithms walk the entire heap to compact live objects towards its beginning.

On average, the minor pause time of the G1 GC is only 71% of the pause time of the ParallelOld GC. In addition, G1 has less spikes in the pause times (e.g., the h2 benchmark has an average pause time of 300ms with the ParallelOld GC, and only 81ms with the G1 GC) . This is mostly due to the fact that G1 can select which regions to collect and consequently can control its pause time as well as the amount of memory that will be freed.

Also, the G1 GC has less and shorter major GCs because a major GC is seen as an emergency that is to be avoided at any cost. The most extreme example is the tradebeans benchmark with a major GC pause time of 590ms in the ParallelOldGC and no major collection in the G1 GC.

Long GC pauses impede the application's availablity. For example, UI applications will need to react within at most half a second on user-input so that the user is not hindered. Similarly, server applications also must react within some time interval to client requests. Long GC pauses will effectively freeze applications. Thus, more concurrent GC algorithms emerge trying to minimize GC pauses, some even try to guarantee a maximum pause time. Benchmarks with long GC pauses are ideal to test these algorithms.

*Young Generation Ratio.*

The young generation ratio shows the maximum amount of memory before a collection in the young generation in relation to the maximum amount of memory before a collection in the old generation. A value below 100% indicates that the young generation is only a fraction of the old generation, whereas a value larger than 100% indicates that the young generation was bigger. This ratio shows us, whether the GC is able to handle most objects as young objects or if it has to keep a lot of them in the old generation.

Please note, that scimark.fft.large has no such ratio, as there was no collection and the VM consequently did not have the chance to adjust the generation sizes. Also, benchmarks without a major collection might have not yet reached the full capacity of the old generation. Consequently, the ratio is a *high estimate* in those cases.

It is interesting to see, that the ParallelOld GC and the G1 GC do not always agree on what generation is dominant (i.e., whether the young generation ratio is below or above 100%). In general, the young generation is bigger in the G1 GC. Thus, the G1 GC keeps potentially dead objects longer in the young generation compared to the ParallelOld GC.

This metric tells us what part of the garbage collection algorithms are under more pressure and enables focused testing and debugging for new collection algorithms.

## 3.4 Object References

Figure 8 examines the object pointers that were recorded by AntTracks during garbage collections. Since AntTracks records pointers only during a collection, the recorded pointers depend on the number of garbage collections (initially introduced in Figure 7). While usually the average number of pointers per object is monadic, DaCapo xalan and luindex, DaCapo Scala factorie and kiama, as well as SPECjvm serial stand out in terms of this metric. All these benchmarks make use of few, but very large arrays to store objects. This results in a high average pointer ratio. The DaCapo database h2 is especially useful for all kind of pointer-related measurements due to its high amount of pointers.

# 4. RELATED WORK

In 2006, Backburn et al. released the DaCapo benchmark suite. At the same time, they published an analysis paper (Blackburn et al. [2]), where they compared their benchmark suite against SPECjvm98 and a modified version of SPECjbb2000. For doing so, they evaluated the DaCapo benchmark suite across different architectures and JVMs. However, 10 years later, Java, JVMs and their components have changed significantly. When they published their paper, the used the Jikes RVM version 2.4.4+. In version 2.9 the Jikes RVM has been substantially rewritten to support Java SE 5. Moreover some benchmarks of the initial release have been completely removed in 2009. New benchmarks, such as avrora, h2, sunflow, are therefore not covered in the paper at all.

In 2006, the DaCapo Scala benchmark suite has been released. In 2011, Sewe et al. [7] analyzed this suite, with a primary focus on the design of the applications. Moreover, they demonstrated the main difference between Java

| | Benchmark | Object Pointers (ParallelOld GC) GCs [#] | Ptrs / GC [#] | Avg ptrs / object [#] | Null rate [%] | Old-to-young rate [%] |
|---|---|---|---|---|---|---|
| DaCapo | avrora | 68.4 | 157,143.4 | 5.1 | 12.7 | 6.4 |
| | batik | 10.0 | 161,875.6 | 4.3 | 43.1 | 3.8 |
| | eclipse | 1.0 | 77,906.3 | 3.0 | 40.5 | 4.8 |
| | fop | 10.0 | 351,025.9 | 3.5 | 45.4 | 1.0 |
| | h2 | 27.6 | 37,958,744.0 | 5.8 | 28.8 | 8.9 |
| | jython | 468.4 | 1,674.8 | 3.8 | 34.6 | 15.9 |
| | luindex | 11.0 | 83,738.7 | 14.8 | 79.1 | 5.4 |
| | lusearch | 7,041.0 | 11,497.7 | 9.0 | 79.0 | 9.1 |
| | pmd | 50.4 | 443,487.8 | 3.6 | 44.9 | 5.5 |
| | sunflow | 774.5 | 80,871.7 | 3.3 | 45.0 | 15.4 |
| | tomcat | 1,234.3 | 33,188.4 | 4.9 | 67.2 | 15.0 |
| | tradebeans | 179.2 | 4,809,331.3 | 6.9 | 50.1 | 6.7 |
| | tradesoap | 973.2 | 885,545.1 | 8.3 | 54.3 | 6.9 |
| | xalan | 5,028.0 | 339,669.5 | 26.3 | 93.2 | 4.3 |
| | actors | 387.5 | 9,185.8 | 2.7 | 25.9 | 13.8 |
| | apparat | 259.8 | 324,520.0 | 2.4 | 23.5 | 8.1 |
| | factorie | 271.9 | 10,287,986.8 | 27.8 | 59.0 | 9.5 |
| | kiama | 28.5 | 822,980.2 | 13.0 | 49.5 | 6.2 |
| | scalac | 45.9 | 1,623,276.1 | 5.5 | 31.3 | 6.9 |
| | scaladoc | 53.0 | 1,133,381.0 | 6.1 | 40.7 | 6.5 |
| | scalap | 40.3 | 87,220.5 | 2.8 | 33.9 | 6.7 |
| | scalariform | 127.8 | 149,652.0 | 2.3 | 8.9 | 13.3 |
| | scalaxb | 48.5 | 189,187.9 | 1.5 | 8.8 | 0.2 |
| | tmt | 2,325.5 | 7,510.2 | 3.7 | 24.5 | 9.8 |
| SPECjvm2008 | compiler.compiler | 165.6 | 8,521,325.4 | 3.4 | 28.5 | 12.2 |
| | compiler.sunflow | 665.9 | 5,843,973.8 | 3.3 | 27.7 | 10.1 |
| | compress | 14.1 | 1,290.0 | 3.2 | 43.2 | 4.3 |
| | crypto.aes | 165.6 | 2,083.5 | 2.9 | 45.3 | 3.1 |
| | crypto.rsa | 1,048.6 | 1,069.7 | 2.5 | 40.2 | 6.9 |
| | crypto.signverify | 752.5 | 5,589.5 | 2.7 | 44.1 | 1.0 |
| | derby | 181.0 | 445,557.6 | 8.2 | 15.1 | 13.6 |
| | mpegaudio | 691.0 | 1,776.3 | 2.2 | 38.1 | 5.0 |
| | scimark.fft.large | 0.0 | - | | | |
| | scimark.fft.small | 322.6 | 240.7 | 2.1 | 46.8 | 19.8 |
| | scimark.lu.large | 2.0 | 24,170.5 | 1.3 | 24.5 | 0.2 |
| | scimark.lu.small | 2,116.4 | 24,004.2 | 2.4 | 38.9 | 0.4 |
| | scimark.monte_carlo | 3.0 | 1,293.0 | 3.3 | 46.2 | 7.3 |
| | scimark.sor.large | 2.0 | 16,820.6 | 1.8 | 42.9 | 0.4 |
| | scimark.sor.small | 3.0 | 3,231.7 | 1.9 | 25.6 | 4.4 |
| | scimark.sparse.large | 2.8 | 906.1 | 3.6 | 51.0 | 5.4 |
| | scimark.sparse.small | 71.8 | 3,257.0 | 2.7 | 45.7 | 1.0 |
| | serial | 384.4 | 51,066.1 | 10.6 | 82.8 | 6.3 |
| | sunflow | 5,540.5 | 88,540.0 | 4.2 | 34.5 | 15.6 |
| | xml.transform | 571.3 | 138,410.2 | 4.2 | 54.4 | 4.4 |
| | xml.validation | 638.4 | 1,201,517.9 | 3.8 | 41.8 | 0.9 |

**Figure 8: Object pointers per GC of all benchmarks**

and Scala code. However, they did not provide information regarding memory, pointer, or garbage collection behavior.

The SPECjvm2008 benchmark suite has been evaluated in 2009 by Shiv et al. [8]. While on the one hand they provide a detailed description of all benchmarks, they also present performance numbers for different hardware and software setups. They used Sun's Hotspot JVM (Java SE 6) to perform their evaluations. Though they do not explicitly state the kind of collector they used, we assume from their description that they already used the ParallelGC. While they also present numbers for allocations and garbage collections, their evaluation focuses on hardware-related performance issues, e.g., performance counters on processors, the effect of SMT (simultaneous multi threading). Therefore, their evaluation does not provide insights into the applications' memory behavior, e.g., in terms of allocations, pointers and GC impact.

Dieckmann et al. [3] published a study on the SPECjvm98 benchmark suite in 1998. For their evaluation they used Sun's Hotspot JVM (Java SE 5) and a tracing algorithm to log all memory-related information. However, their monitoring tool performs additional garbage collections and therefore clearly distorts memory behavior. Apart from that, they provide a detailed analysis of heap sizes, object lifetimes and object layout, as well as object references.

# 5. CONCLUSION

In this paper, we described the memory characteristics and the GC behavior of common Java benchmark suites, i.e., the DaCapo benchmark suite, the DaCapo Scala benchmark suite and the SPECjvm2008 benchmark suite. We showed what benchmarks are best suited when looking for benchmarks with a large total amount of allocations, a high allocation rate, many large objects and large arrays, a high GC count, a high overall GC time, high GC pauses, many young or many old objects, as well as a high pointer density per object. We also showed some curiosities, for example that even though all benchmarks have been warmed up according to the instructions of the respective publishers, the scimark benchmarks allocate most of their objects in code that has not been fully compiled yet, and that the lusearch and pmd benchmarks steer a large part of their normal control flow via exceptions. We want to emphasize that we do not encourage cherry picking benchmarks. However, both researchers and reviewers need to understand whether a selected benchmark meets the characteristics one is trying to test or benchmark.

All tests were applied under modern, state-of-the-art processors, virtual machines, garbage collection algorithms and compilers. We based our analysis on AntTracks, a memory monitoring tool which aims to not distort memory behavior by using a lightweight, VM-internal logging approach.

By revealing internals and curiosities of commonly used benchmarks, this paper provides a basis for explaining outliers in measurement. It can be further used for selecting benchmarks with specific memory characteristics.

# 6. ACKNOWLEDGEMENTS

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

# References

[1] V. Bitto and P. Lengauer. Building custom, efficient, and accurate memory monitoring tools for java applications. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 321–324, New York, NY, USA, 2016. ACM.

[2] 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.

[3] S. Dieckmann and U. Hoelzle. A study of the allocation behavior of the specjvm98 java benchmarks. Technical report, Santa Barbara, CA, USA, 1998.

[4] 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.

[5] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 367–384, New York, NY, USA, 2008. ACM.

[6] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering*, ICPE '15, pages 51–62, 2015.

[7] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 657–676, 2011.

[8] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

# APPENDIX

In this section, we provide additional data we did not discuss in the main sections. Figure 9 and Figure 10 show the same experiment as discussed in Section 3.3, but with a fixed 1GB heap size and an unlimited heap respectively.

| | Benchmark | ParallelOld GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] | G1 GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DaCapo | avrora | 1 | 0 | 0.0 | - | 6 | - | 9,597 | 13 | 0 | 0.4 | - | 2 | - | 223 |
| | batik | 0 | 0 | - | - | - | - | 147 | 0 | 0 | - | - | - | - | 131 |
| | eclipse | 0 | 0 | - | - | - | - | 771 | 2 | 0 | 3.4 | - | 2 | - | 398 |
| | fop | 0 | 0 | - | - | - | - | 2,891 | 3 | 0 | 6.8 | - | 3 | - | 545 |
| | h2 | oom | oom | oom | oom | oom | oom | oom | 296 | 6 | 6.7 | 6.8 | 43 | 21,888 | 16 |
| | jython | 35 | 0 | 1.0 | - | 2 | - | 1,588 | 38 | 0 | 1.0 | - | 2 | - | 1,839 |
| | luindex | 0 | 0 | - | - | - | - | 604 | 1 | 0 | 1.0 | - | 3 | - | 184 |
| | lusearch | 31 | 0 | 3.6 | - | 1 | - | 16,592 | 33 | 0 | 4.1 | - | 1 | - | 20,528 |
| | pmd | 3 | 0 | 3.7 | - | 14 | - | 1,448 | 7 | 0 | 3.2 | - | 5 | - | 622 |
| | sunflow | 18 | 0 | 1.1 | - | 2 | - | 6,593 | 34 | 0 | 1.8 | - | 2 | - | 2,458 |
| | tomcat | 55 | 0 | 0.4 | - | 2 | - | 2,045 | 49 | 0 | 0.4 | - | 2 | - | 3,821 |
| | tradebeans | 138 | 0 | 11.8 | - | 54 | - | 54 | 92 | 0 | 3.2 | - | 21 | - | 149 |
| | tradesoap | 301 | 0 | 45.0 | - | 82 | - | 123 | 216 | 0 | 25.1 | - | 53 | - | 125 |
| | xalan | 27 | 0 | 3.0 | - | 3 | - | 8,715 | 37 | 0 | 2.3 | - | 2 | - | 3,041 |
| DaCapo Scala | actors | 35 | 0 | 0.2 | - | 1 | - | 6,058 | 72 | 0 | 0.4 | - | 0 | - | 342 |
| | apparat | 50 | 0 | 1.1 | - | 19 | - | 519 | 50 | 0 | 0.5 | - | 8 | - | 396 |
| | factorie | 421 | 0 | 51.9 | - | 256 | - | 117 | 267 | 1 | 1.4 | 0.8 | 6 | 978 | 66 |
| | kiama | 1 | 0 | 3.9 | - | 9 | - | 581 | 2 | 0 | 2.4 | - | 3 | - | 504 |
| | scalac | 6 | 0 | 9.9 | - | 41 | - | 10,076 | 10 | 0 | 4.4 | - | 11 | - | 885 |
| | scaladoc | 8 | 0 | 13.9 | - | 3 | - | 10,891 | 10 | 0 | 5.1 | - | 10 | - | 986 |
| | scalap | 0 | 0 | - | - | - | - | 5,815 | 4 | 0 | 3.2 | - | 1 | - | 2,210 |
| | scalariform | 6 | 0 | 0.9 | - | 2 | - | 11,451 | 16 | 0 | 2.4 | - | 3 | - | 12,931 |
| | scalaxb | 13 | 0 | 1.1 | - | 10 | - | 14,787 | 23 | 0 | 0.9 | - | 4 | - | 219 |
| | tmt | 312 | 0 | 0.8 | - | 1 | - | 378 | 495 | 0 | 1.4 | - | 1 | - | 456 |
| SPECjvm2008 | compiler.compiler | 96 | 5 | 51.2 | 6.5 | 115 | 284 | 41 | 57 | 0 | 27.0 | - | 77 | - | 126 |
| | compiler.sunflow | 226 | 7 | 41.3 | 3.0 | 78 | 183 | 45 | 97 | 0 | 10.5 | - | 31 | - | 152 |
| | compress | 2 | 0 | 0.0 | - | 1 | - | 278 | 4 | 0 | 0.0 | - | 1 | - | 177 |
| | crypto.aes | 9 | 0 | 0.2 | - | 2 | - | 1,587 | 7 | 0 | 1.4 | - | 16 | - | 1 |
| | crypto.rsa | 22 | 0 | 0.1 | - | 0 | - | 2,916 | 718 | 0 | 2.5 | - | 0 | - | 150 |
| | crypto.signverify | 24 | 0 | 0.4 | - | 1 | - | 2,886 | 31 | 0 | 3.1 | - | 8 | - | 11 |
| | derby | 220 | 0 | 3.8 | - | 5 | - | 78 | 144 | 0 | 1.6 | - | 2 | - | 133 |
| | mpegaudio | 14 | 0 | 0.1 | - | 0 | - | 3,911 | 98 | 0 | 0.8 | - | 0 | - | 479 |
| | scimark.fft.large | 1 | 1 | 0.5 | 0.2 | 47 | 18 | 71 | 1 | 0 | 0.0 | - | 0 | - | 1 |
| | scimark.fft.small | 18 | 0 | 0.2 | - | 1 | - | 2,373 | 9 | 0 | 2.0 | - | 15 | - | 1 |
| | scimark.lu.large | 4 | 0 | 0.2 | - | 24 | - | 65 | 15 | 0 | 0.3 | - | 7 | - | 24 |
| | scimark.lu.small | 20 | 0 | 0.6 | - | 1 | - | 2,529 | 35 | 0 | 1.1 | - | 1 | - | 1,226 |
| | scimark.monte_carlo | 0 | 0 | - | - | - | - | 1,337 | 2 | 0 | 0.0 | - | 1 | - | 144 |
| | scimark.sor.large | 2 | 0 | 0.5 | - | 20 | - | 187 | 12 | 0 | 0.7 | - | 4 | - | 42 |
| | scimark.sor.small | 0 | 0 | - | - | - | - | 1,409 | 5 | 0 | 0.0 | - | 1 | - | 82 |
| | scimark.sparse.large | 4 | 0 | 0.5 | - | 16 | - | 90 | 1 | 0 | 0.0 | - | 1 | - | 1 |
| | scimark.sparse.small | 1 | 0 | - | - | 0 | - | 7,780 | 42 | 0 | 1.6 | - | 1 | - | 80 |
| | serial | 387 | 0 | 0.9 | - | 1 | - | 125 | 955 | 0 | 1.4 | - | 0 | - | 56 |
| | sunflow | 274 | 1 | 5.1 | 0.0 | 10 | 21 | 49 | 363 | 0 | 2.0 | - | 3 | - | 2,776 |
| | xml.transform | 42 | 0 | 2.3 | - | 3 | - | 209 | 49 | 0 | 2.0 | - | 2 | - | 1,867 |
| | xml.validation | 136 | 4 | 22.4 | 2.6 | 34 | 139 | 48 | 71 | 0 | 10.6 | - | 31 | - | 138 |

**Figure 9: GC count, GC time relative to the total run time (green: less than 5%, yellow: less than 15%, red: more than 15%), and average pause time for minor (-) and major (+) GCs for the ParallelOld GC and the G1 GC respectively (green: less than 10ms, yellow: less than 100ms, red: more than 100ms) (heap limited to 1GB)**

| | Benchmark | ParallelOld GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] | G1 GC Count [#] - | + | Time [%] - | + | Pause [ms] - | + | YR [%] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DaCapo | avrora | 2 | 0 | 0.1 | - | 6 | - | 7,251 | 37 | 0 | 0.7 | - | 1 | - | 179 |
| | batik | 0 | 0 | - | - | - | - | 120 | 0 | 0 | - | - | - | - | 113 |
| | eclipse | 0 | 0 | - | - | - | - | 719 | 2 | 0 | 4.0 | - | 2 | - | 406 |
| | fop | 0 | 0 | - | - | - | - | 2,828 | 3 | 0 | 6.7 | - | 3 | - | 772 |
| | h2 | 8 | 0 | 1.7 | - | 353 | - | 264 | 16 | 0 | 1.1 | - | 116 | - | 112 |
| | jython | 20 | 0 | 0.6 | - | 2 | - | 2,457 | 31 | 0 | 0.9 | - | 2 | - | 2,453 |
| | luindex | 0 | 0 | - | - | - | - | 539 | 2 | 0 | 2.1 | - | 3 | - | 245 |
| | lusearch | 10 | 0 | 2.7 | - | 2 | - | 38,857 | 37 | 0 | 3.4 | - | 1 | - | 15,363 |
| | pmd | 3 | 0 | 3.6 | - | 14 | - | 1,409 | 5 | 0 | 2.4 | - | 6 | - | 1,336 |
| | sunflow | 15 | 0 | 1.0 | - | 2 | - | 8,567 | 26 | 0 | 1.4 | - | 2 | - | 4,697 |
| | tomcat | 61 | 0 | 0.5 | - | 2 | - | 1,721 | 14 | 0 | 0.2 | - | 5 | - | 29,220 |
| | tradebeans | 10 | 0 | 3.3 | - | 197 | - | 1,320 | 90 | 0 | 2.8 | - | 20 | - | 152 |
| | tradesoap | 13 | 0 | 41.8 | - | 1,680 | - | 754 | 64 | 0 | 14.6 | - | 193 | - | 61 |
| | xalan | 11 | 0 | 2.3 | - | 6 | - | 25,235 | 21 | 0 | 0.1 | - | 3 | - | 25,056 |
| DaCapo Scala | actors | 44 | 0 | 0.3 | - | 1 | - | 1,680 | 68 | 0 | 0.3 | - | 0 | - | 3,418 |
| | apparat | 22 | 0 | 0.5 | - | 21 | - | 9,099 | 31 | 0 | 0.3 | - | 11 | - | 7,080 |
| | factorie | 17 | 0 | 5.9 | - | 381 | - | 1,787 | 44 | 0 | 1.2 | - | 33 | - | 604 |
| | kiama | 1 | 0 | 3.5 | - | 9 | - | 518 | 2 | 0 | 1.8 | - | 2 | - | 496 |
| | scalac | 3 | 0 | 5.6 | - | 45 | - | 15,515 | 10 | 0 | 4.0 | - | 10 | - | 3,269 |
| | scaladoc | 3 | 0 | 6.3 | - | 41 | - | 21,921 | 7 | 0 | 4.0 | - | 12 | - | 2,699 |
| | scalap | 0 | 0 | - | - | - | - | 5,686 | 3 | 0 | 3.9 | - | 2 | - | 4,169 |
| | scalariform | 8 | 0 | 1.5 | - | 3 | - | 9,271 | 14 | 0 | 1.9 | - | 2 | - | 13,395 |
| | scalaxb | 16 | 0 | 1.3 | - | 10 | - | 2,034 | 19 | 0 | 0.7 | - | 4 | - | 462 |
| | tmt | 271 | 0 | 0.7 | - | 1 | - | 695 | 220 | 0 | 0.7 | - | 1 | - | 1,517 |
| SPECjvm2008 | compiler.compiler | 2 | 0 | 1.9 | - | 93 | - | 9,525 | 11 | 0 | 6.6 | - | 69 | - | 967 |
| | compiler.sunflow | 6 | 0 | 1.2 | - | 51 | - | 14,019 | 23 | 0 | 2.3 | - | 28 | - | 3,877 |
| | compress | 1 | 0 | 0.0 | - | 1 | - | 373 | 4 | 0 | 0.0 | - | 0 | - | 175 |
| | crypto.aes | 3 | 0 | 0.1 | - | 2 | - | 14,930 | 17 | 0 | 0.3 | - | 1 | - | 6,812 |
| | crypto.rsa | 11 | 0 | 0.0 | - | 0 | - | 4,906 | 301 | 0 | 1.4 | - | 0 | - | 381 |
| | crypto.signverify | 10 | 0 | 0.1 | - | 1 | - | 8,820 | 34 | 0 | 0.5 | - | 1 | - | 2,803 |
| | derby | 8 | 0 | 0.6 | - | 16 | - | 2,516 | 33 | 0 | 0.5 | - | 4 | - | 802 |
| | mpegaudio | 7 | 0 | 0.0 | - | 0 | - | 6,605 | 249 | 0 | 1.5 | - | 0 | - | 251 |
| | scimark.fft.large | 0 | 0 | - | - | - | - | 17,221 | 1 | 0 | 0.0 | - | 1 | - | 3 |
| | scimark.fft.small | 7 | 0 | 0.1 | - | 1 | - | 9,910 | 341 | 0 | 2.8 | - | 0 | - | 302 |
| | scimark.lu.large | 0 | 0 | - | - | - | - | 25,817 | 12 | 0 | 0.3 | - | 11 | - | 70 |
| | scimark.lu.small | 7 | 0 | 0.2 | - | 1 | - | 21,669 | 25 | 0 | 0.8 | - | 2 | - | 3,125 |
| | scimark.monte_carlo | 0 | 0 | - | - | - | - | 1,163 | 0 | 0 | - | - | - | - | 0 |
| | scimark.sor.large | 0 | 0 | - | - | - | - | 11,629 | 11 | 0 | 1.4 | - | 9 | - | 22 |
| | scimark.sor.small | 0 | 0 | - | - | - | - | 1,214 | 1 | 0 | 0.0 | - | 3 | - | 438 |
| | scimark.sparse.large | 0 | 0 | - | - | - | - | 13,222 | 3 | 0 | 0.0 | - | 3 | - | 6 |
| | scimark.sparse.small | 1 | 0 | 0.0 | - | 1 | - | 6,805 | 34 | 0 | 1.1 | - | 0 | - | 448 |
| | serial | 263 | 0 | 0.7 | - | 1 | - | 307 | 804 | 0 | 1.1 | - | 0 | - | 94 |
| | sunflow | 81 | 0 | 0.7 | - | 5 | - | 1,570 | 265 | 0 | 1.4 | - | 3 | - | 2,808 |
| | xml.transform | 1 | 0 | 0.1 | - | 5 | - | 43,979 | 18 | 0 | 0.7 | - | 2 | - | 6,853 |
| | xml.validation | 4 | 0 | 1.3 | - | 55 | - | 9,426 | 22 | 0 | 2.4 | - | 21 | - | 1,442 |

**Figure 10: GC count, GC time relative to the total run time (green: less than 5%, yellow: less than 15%, red: more than 15%), and average pause time for minor (-) and major (+) GCs for the ParallelOld GC and the G1 GC respectively (green: less than 10ms, yellow: less than 100ms, red: more than 100ms) (heap unlimited)**