
Ludwig-Maximilians-Universität München

Institut für Informatik

**Bachelor's Thesis**

# Evaluation of JVM Garbage Collectors for CPAchecker

Tobias Maget

August 29, 2024

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Dr. Philipp Wendler

# Abstract

Garbage collection is an essential component of the Java Virtual Machine's automatic memory management, as it frees up memory occupied by objects that are no longer in use. Given that various garbage collectors can influence the performance of a Java application in different ways, understanding them is important. Additionally, each garbage collector can be individually tuned. In this thesis, we evaluate the impact of selecting and tuning different garbage collectors on the performance of the software verification tool CPAchecker. To do so, we benchmarked CPU time, wall time, and peak memory consumption across various sets of verification tasks as well as individual analyses of CPAchecker. In total, we benchmarked 96 different garbage collection configurations in 762 355 verification runs, which accounted for approximately 2 063 days of CPU time. Our evaluation is driven by two main use cases that highlight different aspects of CPAchecker's performance requirements: One use case focuses on evaluating results in a scientific environment, where the main priority is to reduce CPU time. For this use case, we recommend the Serial Garbage Collector, not only for its good CPU time but also for its ability to prevent "out of memory" errors. Our other use case aims to achieve fast wall time and while preserving a low peak memory consumption. Here, we recommend using the Parallel Garbage Collector tuned with `-XX:MinHeapFreeRatio=80`, as it provides competitive CPU time and is still faster than the Serial Garbage Collector and the Garbage First Garbage Collector. At the same time, peak memory usage can be reduced compared to the current default. Overall, we found that GC performance was stable, with CPAchecker showing little variability in performance across different configurations. However, the quantitative effects of different configurations may vary across different analyses of CPAchecker.

# Acknowledgments

First of all, I would like to thank my supervisor, Prof. Dr Dirk Beyer, for providing me the opportunity to write this thesis at the Software and Computational Systems Lab. I would also like to deeply thank my mentor, Dr. Philipp Wendler, for his support, for taking the time for meetings, and for sharing his valuable experience, which I am sure will benefit me in my future journey. Last but not least, I want to thank the Statistical Consulting Unit StaBLab, Department of Statistics, Ludwig-Maximilians-Universität München, Munich, Germany for their valuable statistical consulting and the highly professional and pleasant collaboration.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

In Java, the Java Virtual Machine (JVM) manages memory automatically. This brings considerable benefits [10]: It eliminates issues like dangling pointers and memory leaks, which enhances the safety of applications. Developers can focus less on the complexities of memory management. This leads to faster development cycles and reduced costs. Java's platform independence ensures that applications can run on any system without modification, providing great flexibility in deployment. However, there is a inherent risk: Garbage Collection (GC) is an essential component of the JVM's memory management, as it frees up memory occupied by objects that are no longer in use. The JVM provides multiple GC algorithms, each with its own individual tuning options. Given that GC significantly influences the performance of a Java application [15], understanding the differences between these algorithms and their respective tuning options is crucial. The default settings might not always be the best for specific applications, implying that even with automated memory management, developers need to be familiar with the underlying GC processes to optimize performance effectively.

In this work, we benchmark and optimize GC configurations for executing the software verification framework CPAchecker [3]. Our primary goal is to enhance performance measures such as CPU time, wall time and memory footprint, while also striving for consistency in results. Instead of delving into the technical specifics of the GC algorithms, our approach centers on effective configurations for each garbage collector, with a particular emphasis on their tuning options.

Our evaluation is driven by two main use cases that highlight different aspects of CPAchecker's performance requirements: One use case focuses on achieving results from CPAchecker expeditiously and with minimal memory usage. This scenario prioritizes rapid and economical outcomes, optimizing for wall time and reduced memory footprint. The other use case focuses on evaluating results in a scientific environment. In this context, CPU time is of primary concern. It is commonly used in competitions like SVComp24 [1] and in research papers that assess state-of-the-art tools or algorithms for software verification. This work can serve as a guide for other developers to better understand GC and to effectively tune its parameters.

## 1.1 Related Work

Due to the advantages of automatic memory management, GC is extensively studied. In his work, Jones [10] explains the architectural design of various GC algorithms in detail. He focuses more on the technical insights and less on choosing or adjusting a specific garbage collector. The Oracle Tuning Guide [17] explicitly concentrates on tuning GC, primarily addressing throughput and latency measures, which do not necessarily coincide with our main measures of CPU time and wall time. However, there is a match in the memory footprint measure. Nonetheless, the guide does not provide quantitative statements about the specific effects of tuning options on performance. Cai [5] examines the costs of GC, also with regard to our measures of CPU time and wall time. Yet, his study explicitly excludes tuning. Oaks [15], on the other hand, theoretically addresses the tuning of GC, particularly in relation to the measures of CPU time and wall time. However, his research is limited to selected tuning options and does not make quantitative statements about the impacts of specific parameters.

## 1.2 Structure

The structure of this thesis is organized as follows: Section 2 provides the necessary background on GC. We explain the different types of garbage collectors and define our performance measures of interest: CPU time, wall time, and peak memory consumption. Before we delve specifically into the impact of GC on CPAchecker, we address the selection and tuning of garbage collectors in Section 3 and Section 4 in general. The focus here is on how different configurations could affect our performance measures. This provides a solid foundation for evaluating the behavior of GC for CPAchecker. In Section 5, we evaluate our benchmark results for CPAchecker. We examine whether our theoretical considerations match the results for CPAchecker and assess which garbage collector or tuning options might be most suitable for our use cases. Finally, in Section 7, we present ideas for future work.

# 2 Background

In Java, the process of GC is essential for managing dynamic memory in the heap [10, 15, 17]. At runtime, the JVM typically allocates objects dynamically in the heap, as the heap is a flexible data structure that can store objects of varying sizes and allows random access. The primary function of GC algorithms is to automatically reclaim space from the heap. This process can be broken down into three steps:

1. Identifying Live and Dead Objects

   The first step in the GC process is to distinguish between "live" objects, which the program will potentially access in the future, and "dead" objects, which are unreachable and whose memory can be safely reclaimed.

2. Reclaiming Memory

   Once dead objects are identified, the garbage collector frees up their space. This reclaimed space is then available for allocating new objects.

3. Compacting the Heap

   At its most basic structure, the heap stores objects sequentially. Given that, removing objects of various sizes can create gaps of various sizes, which leads to fragmentation. To address this issue, GC must compact the heap, reorganizing the memory to minimize wasted space.

Especially when it comes to tuning GC, it is necessary to understand when different garbage collectors execute specific GC steps. The GC algorithms can generally be categorized into two main types: "stop-the-world" and "concurrent". Additionally, it is important to differentiate between mutator threads that execute application code and collector threads that perform GC tasks [6].

## 2.1 Stop-the-world Garbage Collectors

Stop-the-world garbage collectors pause the application during the GC process [10, 15, 17]. This mechanism ensures that all mutator threads are suspended, which

allows the collector threads to proceed without any interference from ongoing operations. The application can only be continued once the entire collection cycle has been completed.

The JVM Serial Garbage Collector (SerialGC) represents the most basic type of stop-the-world garbage collector: Originally, it was designed for simpler computing environments with only one CPU. Therefore, even in multi-processor environments, SerialGC uses only one single thread for collection work.

With the Parallel Garbage Collector (ParallelGC), the JVM offers another GC that implements the stop-the-world approach. Unlike SerialGC, it employs multiple threads to manage GC, which makes it more suitable for multi-processor environments.

## 2.2 Concurrent Garbage Collectors

Concurrent garbage collectors perform at least some of the collection work concurrently, without stopping the mutator threads [10, 15, 17]. However, the extent of concurrency varies among different collectors. In the JVM, there are three concurrent garbage collectors available:

Garbage-First Garbage Collector (G1GC) operates as a mostly concurrent collector. It performs parts of identifying live objects concurrently. Nevertheless, operations such as freeing up memory and compacting the heap still require stop-the-world pauses, which interrupts the application.

In contrast, the Z Garbage Collector (ZGC) and Shenandoah are described as fully concurrent collectors. They are designed to perform almost all GC tasks simultaneously with the execution of application logic. Despite their classification, it is important to note that both ZGC and Shenandoah do incorporate brief stop-the-world pauses, though these pauses do not significantly impact overall application performance [7, 12].

## 2.3 Generational Garbage Collection

Generational garbage collectors operate based on the weak generational hypothesis, which states that most objects are short-lived [17].

In general, the generational design divides the heap into two distinct segments: The young and the old generation [15]. The young generation is further subdivided into eden and survivor spaces. New objects are initially allocated in the eden space within the young generation. When the eden space becomes full, a minor or young GC event occurs: Dead objects are discarded, and those that survive are promoted to one of the survivor spaces within the young generation or eventually to the old generation after surviving multiple young GC cycles. As the old generation reaches capacity, a major or full GC is required to collect the entire heap.

This structure has several advantages [10, 15]: Due to the short lifespan of

most objects, they are typically discarded during minor collections. Consequently, the old generation fills up more slowly, which leads to less frequent full collections. Since the young generation is just a part of the heap, young GC is more efficient than full GC, which requires processing the whole heap. In addition, when a young GC occurs, all objects from the eden space are either discarded or moved to another space, which effectively compacts that portion of the heap. Another major performance advantage is that long-lived objects that are moved to the old generation are only processed during full collections, which happen less often. Even though these major collections are rare, they use a lot of resources because they involve processing and compacting the entire heap.

The implementation of generations varies across different garbage collectors [15, 17]: SerialGC and ParallelGC divide the heap's memory areas into generations sequentially [17]. Both collectors compact the old generation completely during a full GC, which requires a stop-the-world pause. G1GC divides the heap by default into approximately 2.048 distinct regions while still using the same generational approach. G1GC's concurrent part only includes the marking of live objects in the old generation. The collector is termed "Garbage First" because it prioritizes old regions that are mostly garbage. Instead of initiating a full GC, G1GC clears these regions during regular young collections, which is called a mixed GC. It manages objects in the old generation by moving them from one region to another or discarding them, which also helps to compact the heap. Consequently, G1GC generally avoids expensive full GCs, although if the old generation becomes overloaded, a full GC becomes unavoidable.

ZGC is generational since JDK 21 [11], Shenandoah is not a generational garbage collector [8].

## 2.4 Overview of JVM Garbage Collectors

Table 1 provides an overview of the GC algorithms considered in this thesis. The JVM also offers another garbage collector: Epsilon Garbage Collector (EpsilonGC) [19]. However, it performs no memory reclamation, which means it is impractical for our use cases. Additionally, using EpsilonGC incurs certain costs [5], which makes it unsuitable as a basis for determining the costs of other garbage collectors. Consequently, EpsilonGC will not be further considered in the following discussion. The Concurrent Mark Sweep (CMS) Garbage Collector was deprecated in JDK 9 to prioritize the development of other garbage collectors [14]. It was subsequently removed in JDK 14 [18]. As CMS is no longer supported, it will not be considered in this thesis.

Table 1: Overview of Garbage Collectors in the JVM

| Garbage Collector | Operating Principle | Multiple Collection Threads | Generational |
|---|---|---|---|
| **SerialGC** | Stop-the-world | No | Yes |
| **ParallelGC** | Stop-the-world | Yes | Yes |
| **G1GC** | Mostly concurrent | Yes | Yes |
| **ZGC** | Fully concurrent | Yes | Since JDK 21 |
| **Shenandoah** | Fully concurrent | Yes | No |

## 2.5 CPAchecker

CPAchecker[1] is a framework for Configurable Software Verification [2, 3, 20]. It is available as an open-source tool under the Apache 2.0 license. The framework uses the Eclipse CDT Parser to interpret C programs. Being developed in Java, CPAchecker operates within the JVM. The tool was designed to easily integrate verification components of different abstract verification domains. Each of these components must implement the interface of Configurable Program Analysis (CPA).

When a program is submitted to CPAchecker for analysis, it is parsed to construct a set of control flow automata that represent the program. The CPA algorithm within CPAchecker then conducts a reachability analysis based on a CPA. This CPA is provided as an additional input in the form of a configuration and represents the abstract domain that is used for the analysis. As it can be composed of multiple CPAs, this allows for the analysis to be highly flexible. The outcome of this analysis is a set of reachable abstract states, as determined by the CPA algorithm.

## 2.6 Evaluation Measures

For our use cases, GC is optimized in terms of CPU time, wall time and memory footprint, while also striving for consistency in results.

To benchmark CPAchecker, we use BenchExec[2], an open-source framework for reliable benchmarking under the Apache 2.0 license. Accordingly, our evaluation follows the measurement definitions set by BenchExec [4]. In particular, the use of control groups of the Linux kernel allows us to accurately measure resource consumption. CPU time is measured as the total CPU time of the entire process, including all child processes. Wall time is defined as the elapsed time between

[1] `https://github.com/sosy-lab/cpachecker`
[2] `https://github.com/sosy-lab/benchexec`

start and end of a tool execution. We focus on the peak memory consumption of a process, which refers to the minimum amount of resources needed to run the tool and achieve the same results. It reflects the overall memory provided by the operating system. CPAchecker may use various SMT and SAT solvers, some of which execute as native code, separate from the JVM's management. The memory usage of these native libraries is also measured. Therefore, the peak memory consumption accounts for CPAchecker, the JVM, and any native libraries involved.

Not every application execution is identical. Due to various factors, the behavior of GC can be somewhat unpredictable. For instance, small timing differences or varying memory allocations can significantly impact whether an expensive full GC is triggered or not. This introduces a certain level of variability in the performance of the garbage collector. However, it is possible that different GC algorithms vary in how much they are affected by such random influences. A GC algorithm that is less susceptible to these influences would be desirable, as it would likely offer more consistent performance.

In the literature, the measures of throughput, latency, and memory footprint are commonly discussed, while CPU time and wall time receive less attention. Throughput and latency can be defined as follows [21]: Throughput measures the amount of work an application completes within a specific period of time. Latency refers to the application's response time to a request for data. Whether these measures overlap with those used in this thesis is considered in more detail in Section 4.5 and Section 4.6.

# 3 Theoretical Considerations for Selecting Garbage Collectors

Before considering tuning options, we must first discuss the characteristics that GC algorithms offer in their default configurations. These considerations provide the foundation for effective tuning. Even when using the default settings, it may be worth choosing a different garbage collector depending on our measures CPU time, wall time, peak memory consumption and performance variability.

## 3.1 CPU time

In terms of CPU cycles consumed, the architecture of stop-the-world collectors may be beneficial. Pausing all mutator threads provides a consistent snapshot of the heap [10]. This simplifies the task for collector threads, as they can identify live objects without the complication of objects being moved or modified during the collection process. Furthermore, there is no need for synchronization with allocator threads that seek to allocate space.

We hypothesize that the generational design is also advantageous. Processing less old objects consumes fewer CPU cycles [15]. The ability to avoid expensive full GC, as described in section 2.3, should also reduce CPU usage. However, it is important to consider whether more frequent collections of young objects might end up using more CPU cycles. More details on this trade-off are provided in section 4.2.

According to the Oracle Tuning Guide [17], SerialGC is recommended only for single-core environments due to its lack of parallelization. In terms of CPU usage, though, this is its advantage, because parallelization comes with costs [10]. The communication between several collector threads causes CPU overhead. Since SerialGC uses only one collection thread, it can avoid this overhead. Hence, it is likely that SerialGC also performs well in terms of CPU time in multi-processor environments. Cai [5] confirmed this with his study, which examined the CPU usage of all garbage collectors. He found that, across 16 benchmarks and various heap sizes, SerialGC consistently used the fewest CPU cycles on average. On the other hand, there were a few benchmarks in which ParallelGC and one benchmark in which ZGC individually achieved better CPU performance.

Although they use expensive parallelization of work, both ParallelGC and the G1GC retain the advantages of stop-the-world and generational designs. Additionally, G1GC has another benefit: It can avoid costly full GCs by using mixed collections [15]. Although this comes with a trade-off: It requires CPU cycles for multiple background threads to process the old generation and determine the region with the most dead objects.

Shenandoah and ZGC are concurrent garbage collectors. Concurrency requires synchronization not only between multiple collector threads, but also between multiple mutators and collectors operating in parallel [10]. This complex coordination increases CPU cycles [10]. Furthermore, both are not generational, though ZGC started supporting generational collection since JDK 21.

In summary, SerialGC is likely the most efficient in terms of CPU time due to its lack of parallelization overhead. Its efficiency is further enhanced by the stop-the-world approach and its generational design.

## 3.2 Wall time

SerialGC uses only one thread for GC. In a unicore environment, this can be advantageous compared to ParallelGC, which employs multiple collection threads [15]. The reason is that parallelization can introduce time overheads [10]. For instance, to maintain data integrity, exclusive access to shared data structures is required, which means that only one thread can access them at any given time, while the others must wait. The more threads involved, the higher this overhead becomes. Nonetheless, the effect of distributing the workload across multiple threads outweighs the overhead [15], which results in SerialGC usually taking significantly longer wall time than ParallelGC in multi-processor systems. The presence of time overhead implies that wall time improvement does not scale linearly with increasing number of collection threads.

Whether ParallelGC or G1GC performs better in terms of wall time may depend on the specific application and the available heap memory. Full GC pauses are usually very long and have a significant impact on wall time [15]. Here, G1GC can shine with its ability to avoid these pauses, especially in scenarios with frequent full collections [15]. In Cai's [5] study, ParallelGC was the fastest GC on average across 16 benchmarks, except for the smallest heap size. A small heap size likely requires more frequent full collections due to limited memory available for the old generation, which can lead to G1GC outperforming ParallelGC.

Unlike ParallelGC, G1GC aims to achieve latency goals, which can require more frequent but shorter young collections that may take longer overall compared to the less frequent young collections of ParallelGC. Especially, If the old generation is highly occupied, G1GC tries to avoid full GCs at all costs [15]. It responds with more frequent young collections to free memory, which may result in longer wall times compared to ParallelGC that performs a longer full collection but fewer young collections. At the same time, as the old generation

is almost completely full, the concurrent collection threads must process many objects [15]. This can cause the mutator threads to be throttled due to limited CPU availability [15], which may extend the program's execution time as well.

This effect is even more noticeable with the fully concurrent collectors Shenandoah and ZGC, where mutator and collection threads perform all their tasks in parallel. Mutator threads can also experience throttling if the collectors release memory more slowly than it is being allocated [5]. In addition, the necessary synchronization between all threads, not just the collection threads, also results in additional time overhead [10].

In summary, to achieve the best wall time in multi-processor environments, the decision will likely come down to choosing between ParallelGC and G1GC. The decisive factor will be the number of full GCs.

## 3.3 Peak Memory Consumption

By default, the initial heap size is not set to the same value as the maximum reserved heap size. This approach could benefit peak memory consumption, as the GC initially tries to operate with less memory and only increases the heap size when necessary. The specifics of this behavior depend on the heuristics employed by the selected garbage collector [17]. The specifics of how these heuristics can be modified are matters of tuning, which will be discussed in detail in section 4.

General statements about the peak memory consumption of different GC algorithms can only be made based on their design. The parallelization of tasks usually requires more memory [10]. For instance, thread-local data structures are established to enable threads to function more independently. This overhead becomes even more significant in concurrent garbage collectors. They require, among other components, additional structures to track and manage live objects during collection phases without stopping the application [10]. In contrast, SerialGC with its simple single-thread implementation may have the lowest memory overhead.

## 3.4 Variability

To a certain extent, the random performance influences, such as small timing differences, are beyond our control. However, the different GC algorithms implement specific heuristics to adapt their behavior at runtime [17]. We expect the behavior of these heuristics to be likely deterministic, meaning that the GCs algorithms should react similarly in identical situations. Nevertheless, these adjustments are made on the basis of previously gathered experience [15], which includes these random influences. We hypothesize that a GC algorithm that implements fewer heuristics and is thus less sensitive to program execution will be less impacted by random influences on performance.

SerialGC, ParallelGC and G1GC all adjust the size of the generations [17].

While SerialGC only applies the generational design, ParallelGC and G1GC use more advanced strategies: ParallelGC focuses on maximizing throughput by default, whereas G1GC aims to achieve a maximum pause time of 200 ms. To meet these goals, both collectors dynamically shrink and extend the generations during the application's execution [15]. G1GC also adaptively decides the best time to begin concurrent processing and its extent. The implementation of Shenandoah and ZGC are even more complex to achieve extremely short pause times [7, 12]. Since SerialGC implements the fewest and simplest heuristics, we assume that it should deliver the most consistent performance.

# 4 Theoretical Considerations for Tuning Garbage Collection

Having gathered insights into the default behaviors of each garbage collector, we will now delve into various tuning strategies in detail. The primary focus will again be on our evaluation measures CPU time, wall time, peak memory consumption, and performance variability.

## 4.1 Total and Initial Heap Size

The performance of Garbage Collection is significantly influenced by the total size of the heap [5, 10, 15, 17]. This maximum heap size can be adjusted using the `-Xmx` flag [17]. By default, a quarter of the physical memory is reserved for the heap.

Managing the heap size effectively involves a balance [10, 15]: Too small a heap leads to frequent GC as memory fills up more quickly. Consequently, a larger heap size reduces the frequency of these collections. This has the additional advantage that objects remain in memory for longer, which can boost performance if the program accesses them frequently, as they do not need to be reallocated each time. However, increasing the heap size is not without its challenges. Larger heaps mean that when GC occurs, it involves longer pauses because there are more objects to process, though these pauses happen less frequently. The Oracle Tuning Guide [17] suggests that a larger heap size is generally advantageous. Cai's [5] results also clearly show that a larger total heap leads to reduced CPU cycles and wall time.

It is important to be aware of two risks associated with setting the heap size too high. It is crucial that the entire heap is located in physical memory rather than virtual memory that uses disk storage [15]. Using disk storage for the heap can drastically reduce performance because accessing disk storage is far slower than accessing physical memory. This is particularly important during a full GC cycle, when the entire heap is accessed. Secondly, compressed ordinary object pointers reduce the memory usage of object references, but only for heaps up to about 32 GB [15, 16]. If this limit is exceeded, the JVM switches from 32-bit to 64-bit object pointers, which requires additional memory and can neutralize the

benefits of setting a larger heap.

The initial heap size is specified via the `-Xms` parameter[17]. By default, it is set to 1/64th of the physical memory. Every GC algorithm dynamically adjusts the size of this initial heap based on its built-in heuristics, as detailed in Section 3.4. We already discussed that this dynamic adjustment may be advantageous in terms of peak memory consumption, as the initial heap is only expanded when necessary and within the limits by the GC process.

Correspondingly, setting the initial heap size equal to the maximum heap size may consume more memory because the entire heap space is allocated at JVM startup. Though, this could be particularly advantageous in terms of CPU time and wall time and is acceptable if the heap memory is available anyway. Since the entire reserved heap can be allocated first, GCs occur less frequent. However, this leads to the same trade-off as increasing the heap size: GC events will be longer and more costly, as a larger heap must be processed [10, 15]. The consistency of performance may be improved because the adaptive sizing decisions are deactivated when initial heap and total heap are set equal. This could, in turn, negatively impact performance, as these heuristics might have a positive effect on our measures. More details are provided in Section 4.5 and Section 4.6

If the initial heap size and the maximum heap size are not set to the same size, there is another tuning option. It is possible to specify how the heap should be adjusted after each GC event by setting the `-XX:MinHeapFreeRatio` and `-XX:MaxHeapFreeRatio` flags[17].

MinHeapFreeRatio has a default value of 40 percent. If the percentage of free space in a generation drops below 40 percent, the JVM automatically increases the size of that generation until the free space reaches 40 percent, provided it does not exceed the generation's maximum capacity.

MaxHeapFreeRatio has a default value of 70 percent. If the percentage of free space in a generation exceeds 70 percent, the JVM automatically decreases the size of that generation to ensure that the free space does not surpass 70 percent. The minimum size of that generation cannot be undercut.

One might wonder why it is advantageous to increase the total size of the heap in cases the entire heap is not needed to be allocated with memory, especially since all garbage collectors have the same initial heap size. The answer is that dynamic resizing depends on the total size of the heap [17]. A larger maximum heap size allows the JVM to provide more free space in the generations.

## 4.2 Generation Size

Sizing the generations involves determining the ratio between the young generation and the old generation, as well as the ratio between the eden space and the survivor spaces within the young generation[17].

The size ratio of the young generation to the old generation can be controlled using the `-XX:NewRatio` flag[15, 17]. The size of the young generation is calculated as

$$\frac{1}{1 + \text{NewRatio}}$$

and correspondingly the size of the old generation is

$$1 - \frac{1}{1 + \text{NewRatio}}.$$

For instance, with a default NewRatio of 2, one third of the heap is dedicated to the young generation, and two thirds to the old generation.

The larger the young generation, the less frequent, but more expensive and longer the young collections are, as more objects need to be processed [10, 15]. At the same time, fewer objects are transferred to the old generation because more objects no longer reach the necessary age for the old generation and are discarded beforehand. The risk here is that the old generation is relatively small due to the large young generation, so that it fills up quickly and leads to potentially more full GCs [15]. Generation sizing is therefore strongly dependent on the lifespan of the objects.

The size ratio between the eden space and the two survivor spaces within the young generation is controlled by the `-XX:SurvivorRatio` flag[15, 17]. The size of a survivor space is

$$\frac{1}{2 + \text{SurvivorRatio}}$$

and since there are two survivor spaces, the size of the eden space is calculated as

$$1 - 2 \times \frac{1}{2 + \text{SurvivorRatio}}.$$

With a default value of 8, each survivor space occupies 10 percent of the young generation, with eden taking up the remaining 80 percent.

The aim of the survivor spaces is to prevent objects from being promoted to the old generation after just one young collection [15]. If the survivor spaces are too small, they fill up quickly [15]. If the survivor spaces are full, the JVM promotes objects directly from eden space to the old generation, which can increase full GC events, consuming more CPU cycles and wall time. Consequently, larger survivor spaces can reduce the number of objects promoted to the old generation, which reduces full GC events and can reduce CPU cycles and wall time. However, if they are too large, these spaces may be poorly utilized, which results in

less memory available for eden space and the need to perform smaller collections more frequently [15], which could in turn increase CPU load and wall time. In addition, the efficiency of survivor spaces decreases if many objects are long-lived and thus end up in the old generation in any case [15].

G1 offers additional parameters to fine-tune the size of the young generation: `-XX:G1NewSizePercent` and `-XX:G1MaxNewSizePercent`, with default values of 5 and 60 [17]. These parameters define the minimum and maximum percentage of the heap that can be allocated to the young generation.

Overall, generation tuning is very much dependent on the application itself. It is advisable to collect information about the age distribution of objects in advance using the `-Xlog:gc+age=trace` flag [17]. The age of an object indicates how many GC events it has already survived in the young generation. The age distribution of objects is measured in bytes, as the size of an object is typically represented in bytes.

The size of the generations is generally adjusted adaptively by the GC at runtime. This is done to achieve its internal heuristics, for instance the throughput target [17]. To do so, the GC uses the experience gained so far [15]. The set ratios NewRatio and SurvivorRatio are however not without effect, as they specify the upper limit of the size of the generations within which adaptive adjustments can operate. These limits are also maintained in proportion when the heap expands [15].

The advantage of adaptive sizing includes achieving the GC's heuristic goals, which could have a positive effect on our measures. This is explained in more detail in section 4.5 and section 4.6. Yet, deactivating this feature could be advantageous for our measures. The performance could be more predictable as the size of the heap and generations must be determined in advance. In addition, the calculations required for adaptive sizing introduce overhead [15]. Another disadvantage is that when the program goes through different phases, previously gained experience may become irrelevant and decisions may be made incorrectly [15].

Adaptive sizing can be disabled via the `-XX:-UseAdaptiveSizePolicy` flag. Though, this setting only takes effect if the initial heap size is set to match the total heap size.

## 4.3 Number of Parallel Threads

The number of parallel threads dedicated to GC can be managed using the `-XX:ParallelGCThreads` flag [17]. This flag specifies the number of threads used by ParallelGC and G1GC for collection work during the stop-the-world pauses. It is important to differentiate these from the concurrent threads used by fully concurrent collectors and those used by G1GC to mark live objects of the old generation.

The JVM dynamically determines the number of parallel threads based on the number of cores available at the time of JVM startup [17]. It refers to physical cores if hyperthreading is not enabled, and virtual cores if hyper-threading is enabled. On machines with eight or fewer cores, the JVM assigns one GC thread per core. On machines with more than eight hardware threads, 5/8 of the physical cores are dedicated as GC threads. The result is rounded down to the nearest integer [15].

According to Cai [5], the performance of GC is usually only evaluated on the basis of wall time. To minimize this time, a high level of parallelization is achieved by a high number of parallel threads. This means that the same work can be completed in a shorter time and therefore, a decent wall time is guaranteed by default. However, CPU usage is neglected in this approach. During the stop-the-world pauses, the JVM tries to dedicate all possible CPU resources to the parallel threads [15]. If a smaller number of threads is used and these cannot utilize all the CPU resources, we may expect that CPU cycles are effectively saved. In addition, high parallelization causes further CPU overhead, as described in section 3.1. In conclusion, using many parallel threads reduces wall time. To decrease the consumed CPU cycles, however, we assume it is advisable to reduce the number of threads.

For each additional thread, thread-local data structures must be created [10]. In addition, each thread reserves a portion of the old generation for promotions during young collections [17]. As a result, the old generation is divided into several segments based on the number of threads, which can potentially lead to fragmentation. Therefore, we assume that memory overhead increases with each additional thread.

Although the number of parallel threads is ergonomically determined based on available system resources–if not explicitly set–the logic remains consistent. This means that the same number of parallel threads will be used under similar conditions, without runtime adjustments. Therefore, a chosen configuration should yield consistent results.

## 4.4 Number of Concurrent Threads

The flag `-XX:ConcGCThreads` determines the number of concurrent threads [17]. It is important to distinguish how much work these threads perform depending

on the garbage collector in use. G1GC's concurrent threads primarily handle the marking of live objects in the old generation. In contrast, the concurrent threads of ZGC and Shenandoah perform most of the GC work.

In the case of G1 and Shenandoah, the JVM also dynamically determines the number of concurrent threads based on the number of (virtual) cores available at JVM startup [9, 17]. By default, G1GC uses a quarter of the number of parallel threads as concurrent threads. Shenandoah uses a quarter of the available (virtual) cores as concurrent threads. Both results are rounded down to the nearest integer. In contrast, ZGC employs heuristics to automatically select the number of concurrent threads, which can be overridden by setting the flag [17].

With regard to wall time, it could be that the rule applies again: More threads can do the same work in less time. However, as outlined in section 3.2, too many concurrent threads may throttle mutator threads due to limited CPU availability. Conversely, too few threads can cause the memory to be requested faster by the application than it can be provided by the GC, which leads to another form of mutator thread throttling [5, 15]. In both scenarios, wall time could potentially increase.

Coordination between all concurrent threads consumes significant CPU resources [10]. It could be that a higher number of concurrent threads increases this effort and thus also the CPU usage, while a lower number might result in reduced CPU consumption.

In the case of G1, there is a particular risk that too few concurrent threads will lead to more frequent full collections [15], which could be costly for both wall time and CPU time. This occurs when the old generation fills up faster than the concurrent threads identify the regions with the most dead objects. The JVM then aborts the concurrent marking activity and performs a full collection. As a result, the strong ability of G1GC to avoid full collections is lost.

Except for ZGC, the performance should be consistent because the number of threads is determined once at startup and then maintained throughout the execution.

## 4.5 Throughput Goal

The throughput goal is adjusted by the `-XX:GCTimeRatio` flag and applicable to ParallelGC and G1GC [17]. It is defined as the fraction of time that is not consumed by GC relative to the total execution time of the program [15]. It is calculated as follows:

$$\text{Throughout Goal} = 1 - \frac{1}{1 + \text{GCTimeRatio}}$$

Here, the subtracted term represents the proportion of time that the GC is active. For instance, the default GCTimeRatio for ParallelGC is 99, meaning that the GC consumes 1/100, or 1 percent, of the program's execution time. Consequently,

the actual program code runs for 99 percent of the time. In contrast, G1GC has a default GCTimeRatio of 12, which implies a GC time of about 1/13, or approximately 8 percent of the total execution time.

The higher the total GC time, the lower the throughput. One could conclude that lower throughput leads to a longer wall time as well, as GC takes more time. For ParallelGC, this could be true since collector threads are only active during pauses, and long GC pauses directly extend wall time. In the case of G1GC or other fully concurrent collectors, special care must be taken. Concurrent collector activities can lead to the mutator threads being throttled [5, 15]. If only GC pause times are considered in the calculation of throughput, concurrent collectors may appear to achieve high throughput even though the wall time might actually be longer due to reduced mutator work.

Internally, the JVM prioritizes achieving the throughput goal at the expense of increased memory usage [15]. If the throughput goal is not met, the JVM enlarges the heap size, which implies that a higher throughput could lead to higher peak memory consumption. At the same time, the larger heap could improve wall time and CPU usage, as explained in section 4.1. A high throughput can therefore be beneficial for this measures.

The adaptive resizing of the heap and generations aligns with the set throughput goal, meaning that the specific level of the throughput goal should not change the variability of performance. More consistent performance should only be achieved if the pause time goal is disabled. Setting the total heap size equal to the initial heap size effectively deactivates the throughput goal.

## 4.6 Maximum Pause Time Goal

The maximum pause time goal is set using the `-XX:MaxGCPauseMillis` flag and applicable for ParallelGC and G1GC [17]. This parameter sets the maximum allowable duration, in milliseconds, that any GC pause can last[15]. It is irrelevant whether it is a young or a full collection. The idea behind this parameter is to ensure short pause times, which typically lead to lower latency and better response times.

For G1GC, the default maximum pause time is set at 200 milliseconds [17]. ParallelGC does not have this parameter set by default [17]. This indicates its focus on high throughput rather than latency constraints. In contrast, G1GC tries to balance both throughput and latency goals. The prioritization of goals for both collectors is as follows: Pause time goal takes precedence over throughput goal and then over memory footprint [17].

To achieve the maximum pause time goal, the JVM decreases the size of the heap, which allows collections to occur more quickly since fewer objects need to be processed [15]. G1GC can further activate its concurrent threads earlier and adjust how many regions of the old generation are processed during each GC cycle [15].

In the case of ParallelGC, shorter pauses could contribute to reduced overall wall time. However, reducing the size of the heap results in significantly more frequent collections. It is possible that the shorter pauses will add up to a longer total wall time [15] and CPU time as well. In the case of G1GC, increased concurrent activity may again throttle mutator threads [5, 15], which could to a longer wall time as well.

As the pause time goal is achieved by reducing the heap size, memory usage could decrease significantly. The variability of performance should remain. More consistent performance can only be achieved if the pause time goal is disabled. This is possible by setting the total heap size equal to the initial heap size.

# 5 Experimental Evaluation

In this section, we evaluate the performance of CPAchecker under various GC configurations based on the theoretical considerations in Section 3 and Section 4, applying the performance measures described in Section 2.6. The goal of this evaluation is to identify the most suitable GC configuration for each of our two use cases, which ensures that CPAchecker meets the specific performance requirements of both scenarios. As explained in Section 1, the first use case aims to achieve results from CPAchecker expeditiously with minimal memory usage. Here, wall time and the peak memory consumption are of primary concern. The second use case focuses on evaluating results in a scientific setting, where the main priority is to reduce CPU time. We first benchmarked all five JVM garbage collectors out-of-the-box, without any tuning. Subsequently, the garbage collectors were individually tuned and benchmarked. This approach allows for a comparison of the performance between the default configurations and the tuned versions. Currently, CPAchecker uses G1GC as the default garbage collector. Due to limited resources for benchmarking, only configurations that are beneficial for the use cases will be further pursued. Furthermore, each tuning parameter requires ongoing maintenance because they may become outdated with newer JDK versions. Therefore, our goal is to identify tuning options that deliver a clearly positive impact. Parameters that offer only minimal effects will not be considered further.

## 5.1 Experiment Setup

In total, we benchmarked 96 different garbage collection configurations in 762 355 verification runs, which accounted for approximately 2 063 days of CPU time. Time and peak memory consumption results were rounded to three significant digits.

### 5.1.1 Software

We conducted the experiments using CPAchecker in revision 46761 from the project repository[3]. For benchmarking, we used BenchExec[4] in version 3.21 and executed the benchmarks through BenchCloud[5]. We used Ubuntu 22.04.4 (64-bit) as operating system and CPAchecker was executed with OpenJDK 17.0.11 as JVM. The experiments follow the SV-COMP24 [1] setup, as this competition corresponds to the scientific environment of the second use case. Consequently, the resource limits were set to match those of SVCOMP24: 15 min of CPU time, a hard limit of 16 min, 15 GB of memory, and 4 CPU processing units. However, in Section 5.2.5 we also experimented with 8 CPU processing units to allow for a broader variation in the number of GC threads.

As mentioned in Section 2.6, some analyses delegate verification tasks to native libraries that operate outside the JVM. Since these native libraries also consume memory, the available 15 GB memory must be divided to ensure both the JVM and native libraries have enough memory to avoid crashes [20]. To achieve this, the JVM's total heap size must be set smaller than the available memory, as it is fully reserved at JVM startup. Thus, we set the heap size to 10 GB.

We configured CPAchecker according to SV-COMP24 using the options `-svcomp24` and `-benchmark`. Since our focus is on performance data for CPAchecker, we disabled the output of witness files.

### 5.1.2 Verification Tasks

We used the set of verification tasks from SV-COMP24 [1], which is available in the project repository[6] under the tag `svcomp24-final`.

Due to limited resources, we narrowed down the task set to a subset of tasks. The idea was to first focus on tasks where GC performance is relevant, so we could see the effects of tuning more clearly. At the same time, reducing the number of tasks allowed us to efficiently utilize the available resources to benchmark a variety of GC configurations. To achieve this, the entire set was initially benchmarked with a doubled CPU time limit of 30 min. We used G1GC, as it has been the default up until now. For all tasks we removed the default CPU limit of 15 min in the properties. In addition, for tasks where the property is reachability safety, we set the option `-setprop limits.time.cpu::required=1800s`. The CPU limit for individual components of the tasks remained unchanged, which ensures that the algorithm of each component within a task continues to execute. Only the algorithm of the last component benefited from more CPU time. For the subset, we further considered only the tasks that did not time out within the CPU

[3] `https://svn.sosy-lab.org/software/cpachecker/trunk`
[4] `https://github.com/sosy-lab/benchexec`
[5] `https://vcloud.sosy-lab.org/cpachecker/webclient/master/info`
[6] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`

limit of 30 min. The advantage of this approach is that if a GC configuration improves CPAchecker's performance in terms of CPU time, a task that previously required between 15 and 30 min of CPU time might now complete within 15 min. Consequently, tasks that exceed 30 min of CPU time are not further considered as it is unlikely that they are processed within 15 min. In addition, these tasks require a significant amount of benchmarking resources. Tasks with statuses such as ERROR, OUT OF MEMORY, ASSERTION, EXCEPTION, or UNKNOWN were explicitly included in the evaluation. This is due to the possibility that a different GC configuration could also improve CPAchecker's performance with respect to these tasks, for instance, by detecting an error more quickly. As a second criterion, we excluded tasks where GC time accounted for more than 2 percent of the wall time. CPAchecker provides statistics on GC via MXBeans[7], which are output through the BenchExec logs. The method `long getCollectionTime()` returns the approximate accumulated GC time in milliseconds. This output helps identify tasks that are notably affected by GC. To achieve this, we compared the reported GC time to wall time, since both refer to elapsed time, and selected tasks where GC time accounted for more than 2 percent of the wall time. This restriction allows us to efficiently narrow down the set to tasks where GC performance is relevant. In total, this resulted in a subset of 2 822 verification tasks. This subset still includes tasks from all the original properties, such as reachability safety, no data race, memory cleanup, memory safety, no overflow, and termination.

### 5.1.3 Hardware

For the experiments, two different sets of machines were used to maximize benchmarking resources. It was ensured that only GC configurations executed on the same machine were compared during the evaluation. In addition, the assignment of benchmarks to each machine was clearly documented.

The first set of machines used for the experiments was equipped with an Intel Core i7-10700 processor running at 2.9 GHz and 67 GB of RAM. This processor has 8 physical cores and supports hyperthreading, which provides a total of 16 virtual cores. Turbo Boost was disabled, as it can influence the results by automatically adjusting the CPU frequency [20]. This machine was used to determine the subset of SV-COMP24 verification tasks and to evaluate the performance of GC with these subset tasks.

The second set of machines featured an Intel Xeon E3-1230v5 processor with a frequency of 3.4 GHz and 33 GB of RAM. This CPU has 4 physical cores and also supports hyperthreading, which results in 8 virtual cores. Turbo Boost was disabled on this machine as well. It was used to benchmark the entire SV-COMP24

[7] `https://docs.oracle.com/en/java/javase/11/docs/api/java.management/javax/management/MXBean.html`

verification task set, along with the additional set of tasks for k-induction, predicate analysis, and value analysis.

Both sets of machines ran on the Linux kernel version 5.15. The number of CPU units specified by the resource limit matches the number of virtual cores. All benchmarks were executed in parallel. For instance, when using 4 CPU units on a machine with an Intel Core i7-10700 processor, which has 16 virtual cores, 4 parallel executions were possible, with sufficient RAM available for each execution (15 GB per run). On a machine with an Intel Xeon E3-1230v5 processor, only two parallel executions were possible due to its 8 virtual cores, but there was still enough RAM for each execution (15 GB per run). When using 8 CPU units, the number of parallel executions was halved accordingly. These parallel executions can lead to nondeterministic interference affecting performance [20]. However, this approach was taken to better utilize available resources, similar to the practice in SV-COMP24. Additionally, parallel execution is common when CPAchecker is used for evaluating algorithms rather than performance.

### 5.1.4 Reproduction of Results

All necessary resources for reproducing the results are provided [13]. This including benchmark definitions for BenchExec to rerun the experiments. Additionally, tables with the complete results are available, along with the raw measurement data in BenchExec format.

## 5.2 Experiment Results

In this section, we evaluate the benchmark results. The average CPU time for the entire set of tasks is calculated based on all tasks that did not result in an "out of memory" error. It is important to exclude these errors, as they could occur with a low CPU time. Conversely, a more memory-efficient GC might prevent such errors but require more CPU time for the same task. Timeouts were taken into account with a CPU time value of 900 seconds.

### 5.2.1 Selecting Garbage Collectors

First, we analyzed all five GC algorithms in their default configurations, without applying any tuning parameters. Table 2 provides a summary of the results and Figure 1 presents quantile plots with results for CPU time, wall time, and memory footprint for correct tasks. Table 3 lists the number of different GC cycles for each generational garbage collector for all tasks of the subset.

In terms of CPU time, we observe that running CPAchecker with SerialGC, ParallelGC, and G1GC consumes, on average, fewer CPU cycles than with Shenandoah and ZGC. This results in fewer correct tasks and more timeouts for ZGC and Shenandoah. ParallelGC and G1GC perform quite similarly, and SerialGC

requires noticeably less CPU time for the same set of verification tasks. When comparing wall time, ParallelGC is the fastest, closely followed by G1GC. ZGC and Shenandoah have slower wall times, while SerialGC is noticeably the slowest. The performance differences between these GC configurations become more pronounced with increasingly difficult tasks, as confirmed by Figure 1. Regarding peak memory consumption, running CPAchecker with Shenandoah and particularly ZGC requires a substantial amount of memory, significantly higher than the other garbage collectors. While the available memory is generally sufficient, there is a slightly increased number of "out of memory" errors. G1GC and ParallelGC show similar peak memory consumption, though G1GC tends to use more memory. SerialGC, in contrast, requires the lowest amount of memory.

We observe that SerialGC and ParallelGC perform a significantly higher number of full GCs compared to G1GC, with SerialGC executing slightly more full GCs than ParallelGC. SerialGC also leads with the number of young GCs. G1GC performs more young GCs than ParallelGC but still substantially fewer than SerialGC. Additionally, G1GC performs 6 997 concurrent marking cycles.

Our experimental results confirm that, in terms of CPU time, SerialGC is competitive and even outperforms other collectors in multi-processor environments. Compared to ParallelGC, SerialGC performs more full GCs and significantly more total GCs, yet it still consumes fewer CPU cycles. We conclude that the parallelization overhead becomes evident here. Although ParallelGC performs fewer GCs, they require more CPU resources because the heap is processed in parallel, which necessitates additional synchronization between the collector threads. At the same time, using multiple threads for GC significantly shortens the wall time. We assume that ParallelGC achieves its lower number of GCs by earlier expanding the heap size to meet its throughput goal, which results in a higher peak memory consumption.

We may conclude that G1GC achieves similar CPU usage to ParallelGC due to its fewer expensive full collections. This allows G1GC to compensate for its more frequent collections needed to meet latency goals and the CPU usage of its background threads. In terms of wall time, though, the more frequent collections and the potential mutator throttling due to the CPU consumption of background threads become more relevant factors, making ParallelGC significantly faster. However, it is important to note that in the case of CPAchecker, the available 15 GB of memory is quite generous. If less memory is provided, G1GC's ability to avoid full collections could become more relevant, which could potentially make it more efficient in terms of both CPU time and wall time.

Our results highlight that running CPAchecker with the concurrent collectors Shenandoah and ZGC requires high CPU usage. This is due to the increased parallelization overhead, as the work is done concurrently rather than during stop-the-world pauses. Additionally, these collectors are not generational. Despite using multiple collector threads, they are still slower than G1GC and Paral-

lelGC, which could indicate that mutator threads are being throttled. The high peak memory consumption is likely due to the additional resources required for concurrency.

When considering untuned GC configurations, G1GC and ParallelGC are promising options for our use case, where wall time and peak memory consumption are the primary concerns. For the other use case, which aims to minimize CPU time, SerialGC is preferable due to its strong CPU performance and minimal memory requirements to operate efficiently. ZGC and Shenandoah demand excessive CPU resources, which leads to a reduced number of correct tasks and increased timeouts. While the memory available for benchmarking is sufficient, the higher memory consumption of CPAchecker is not problematic in itself. However, it increases the risk of "out of memory" errors. This risk is even more significant becomes even more significant when using different native libraries that require additional memory. Consequently, we will limit our further research and tuning efforts to SerialGC, ParallelGC and G1GC.

Table 2: Results for each garbage collector in default configuration

| **Garbage Collector** | **Correct Tasks** | **Timeouts** | **Out of memory** | **Avg. CPU time (s)** |
|---|---|---|---|---|
| **SerialGC** | 2 197 | 148 | 0 | 176 |
| **ParallelGC** | 2 193 | 155 | 0 | 183 |
| **G1GC** | 2 198 | 146 | 1 | 184 |
| **Shenandoah** | 2 169 | 179 | 3 | 196 |
| **ZGC** | 2 138 | 214 | 4 | 200 |

Table 3: Number of garbage collections and concurrent marking cycles of live objects for all tasks of the subset for each generational garbage collector

| **Garbage Collector** | **Full GCs** | **Young GCs** | **Concurrent marking cycles** |
|---|---|---|---|
| **SerialGC** | 15 724 | 330 381 | 0 |
| **ParallelGC** | 12 189 | 72 202 | 0 |
| **G1GC** | 166 | 160 278 | 6 997 |

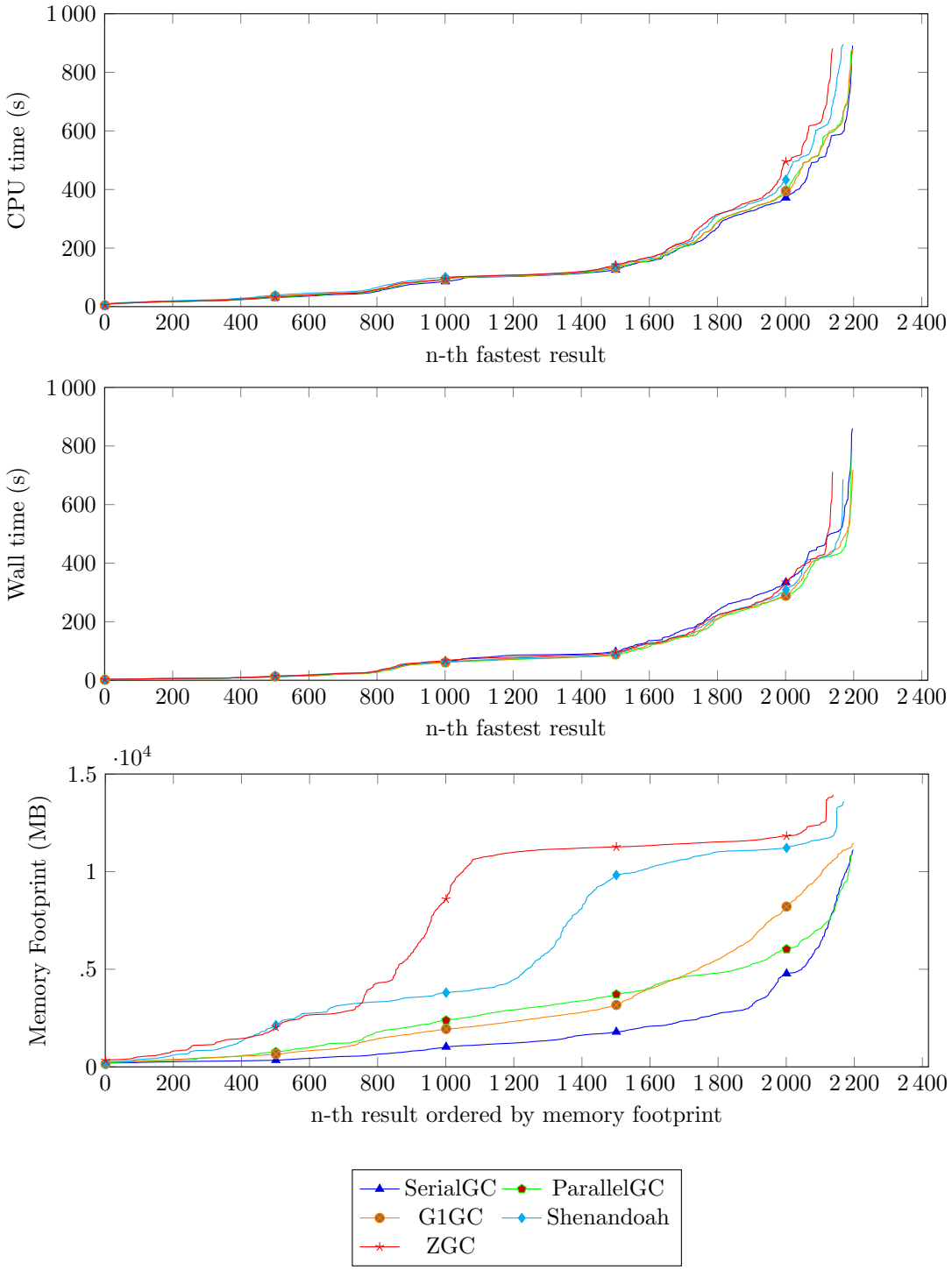



Figure 1: Quantile plots of correct tasks for CPU time, wall time and memory footprint across all garbage collectors in default configuration

### 5.2.2 Overview of Tuning Parameters and their Default Values for CPAchecker

Table 4 summarizes all tuning parameters for SerialGC, ParallelGC, and G1GC, along with their default values specifically for CPAchecker and our experiments. If a field is left blank, the parameter is not available for the respective garbage collector. An exception to this is the `MaxGCPauseMillis` parameter for ParallelGC, which, although available, is not set by default. The default values listed have been sourced from the Oracle Tuning Guide [17] or derived from the JVM logs generated during our experiments.

Table 4: Overview of tuning parameters for SerialGC, ParallelGC and G1GC and their default values for CPAchecker

| Parameter | SerialGC | ParallelGC | G1GC |
|---|---|---|---|
| `Xmx (MB)` | 10 000 | 10 000 | 10 000 |
| `Xms (MB)` | 224 | 224 | 224 |
| `MinHeapFreeRatio` | 40 | 40 | 40 |
| `MaxHeapFreeRatio` | 70 | 70 | 70 |
| `NewRatio` | 2 | 2 | 2 |
| `SurvivorRatio` | 8 | 8 | 8 |
| `G1NewSizePercent` | | | 5 |
| `G1MaxNewSizePercent` | | | 60 |
| `ParallelGCThreads` | | 4 | 4 |
| `ConcGCThreads` | | | 1 |
| `GCTimeRatio` | | 99 | 12 |
| `MaxGCPauseMillis` | | | 200 |

### 5.2.3 Total and Initial Heap Size

In the case of CPAchecker, the total heap size must be determined upfront, as described in Section 5.1.1. Therefore, it is an obvious choice to set the initial heap size equal to the total heap size. To do so, we set `-Xms` to match `-Xmx` at a value of 10 GB. Due to limited benchmarking resources, we did not experiment with varying the heap size itself. In a second step, we experimented with different values for `-XX:MinHeapFreeRatio` and `-XX:MaxHeapFreeRatio`. Table 6 presents a summary of the results, where the average CPU time is calculated based on all tasks that did not result in an "out of memory" error. Table 7 shows the average measures for the commonly solved subset of each tuned and untuned GC variant and Table 5 lists the number of different GC cycles before and after tuning the initial heap size. In Figure 2, line plots illustrate the performance

measures for ParallelGC across various `-XX:MinHeapFreeRatio` settings for the commonly solved subset.

In the following paragraphs, we first discuss setting `-Xms` to match `-Xmx`. We can see that this tuning reduces CPAchecker's CPU usage for both SerialGC and ParallelGC. However, with ParallelGC, this results in fewer correct tasks and more timeouts. Using G1GC, it leads to a higher CPU time and and also to an increase in timeouts. Regarding wall time, only SerialGC shows an improvement when tuned, whereas ParallelGC and G1GC perform worse in their tuned configurations. Additionally, the peak memory consumption increases significantly for all three collectors, with SerialGC requiring more than twice the amount of memory to run CPAchecker. In terms of GC cycles, tuning has led to substantial reductions, with SerialGC and ParallelGC experiencing decreases of approximately 90 percent and 55 percent, respectively. G1GC, while also seeing a reduction, has a relatively smaller decrease of about 35 percent.

Due to the tuning option, the entire heap is now reserved upfront, meaning that larger generations are allocated at the start. This allows CPAchecker to allocate more objects before a GC is required, which results in higher peak memory consumption. At the same time, when memory is freed up, more space is created than with smaller generations, allowing for more objects to be allocated before the next GC is triggered. This leads to less frequent collections. This effect is particularly evident with SerialGC, where the number of collections has decreased significantly, while there has been a substantial increase in the peak memory footprint.

However, we may conclude that less frequent collections do not necessarily improve CPAchecker's performance across all measures. While fewer collections might seem beneficial, they can be more costly because they involve processing a larger portion of the heap and, consequently, a larger number of objects [10, 15]. For instance, in the case of ParallelGC and G1GC, the wall time increases, and although G1GC experiences fewer concurrent cycles, these background cycles will likely consume more CPU cycles as more objects must be processed. We assume this effect is even more pronounced because, for ParallelGC, the throughput goal is deactivated, and for G1GC, the latency goal is turned off. This allows ParallelGC to expand the heap more quickly, as it is not constrained by the overall pause time. Similarly, with the latency goal turned off, G1GC does not prioritize maintaining a smaller heap size.

The reason for the increased number of timeouts with tuned ParallelGC remains unclear. It is possible that the higher initial heap size may have interfered with CPAchecker's analysis itself.

Regarding our use cases, setting `xms=xmx` alone is not effective. While it does improve CPU time for ParallelGC and SerialGC, the benefits are limited. For SerialGC, this setting results in only one fewer timeout, and for ParallelGC, there is a noticeably reduction in the number of correct tasks. For G1GC, there

are no positive effects observed. Additionally, with adaptive sizing turned off, maintaining the configuration becomes more challenging, as the generation size must be predetermined.

Now we consider tuning the heap sizing via `-XX:MinHeapFreeRatio` and `-XX:MaxHeapFreeRatio`. For G1GC, there are no major changes when deviating from the default settings of `-XX:MinHeapFreeRatio=40` and `-XX:MaxHeapFreeRatio=70`. However, with SerialGC and ParallelGC, lowering the `MinHeapFreeRatio` leads to higher average CPU times and more timeouts. For SerialGC, it worsens average wall time, while for ParallelGC, it results in a slight improvement. CPAchecker's peak memory consumption is reduced with SerialGC and even more noticeably with ParallelGC. We observe that increasing the `MinHeapFreeRatio` has the exact opposite effects, except that CPAchecker continues to need less memory with ParallelGC than with its default configuration. Notably, this remains true even when the flag is set to match the default value. For `MaxHeapFreeRatio`, we observe that changes do not affect G1GC. Setting it lower than the default 70 results in higher CPU usage for both SerialGC and ParallelGC, while wall time and peak memory usage are only improved for ParallelGC.

The `MinHeapFreeRatio` determines the minimum percentage of free space in a generation after a GC event has occurred [16]. Thus, we may conclude that lowering this value results in less expansion of the generations and, consequently, the overall heap, which reduces peak memory consumption. However, this typically comes at the cost of increased CPU usage, as the smaller amount of free memory triggers more frequent GC events. Conversely, a higher value leads to more higher peak memory usage but reduces the frequency of GCs, which potentially reduces CPU usage.

The `MaxHeapFreeRatio` determines the maximal percentage of free space in a generation after a GC event has occurred [16]. We assume that a lower value ensures that GC events occur earlier and therefore less memory is used. However, this also implies more frequent GC events at the expense of CPU time. We expect that a higher value could lead to the exact opposite effect.

We hypothesize that the lower peak memory consumption with ParallelGC when the `MinHeapFreeRatio` flag is set, regardless of the specific value, is due to a change in the priority of ParallelGC's heuristics. Normally, throughput is prioritized over memory efficiency [17], but with the flag set, this is no longer the case. This could be why the default behavior of ParallelGC differs from when the flag is set, even if the same value is used. We also assume that this explains the difference between ParallelGC compared to SerialGC and G1GC, as SerialGC does not implement internal goals, while G1GC employs much more complex ones.

The difference in higher CPU usage with a lower MaxHeapFreeRatio compared to a lower `MinHeapFreeRatio`, both of which constrain the generations and

therefore the heap, is due to the fact that freeing the heap is more expensive than expanding it.

For our use cases, setting `MinHeapFreeRatio=80` seems promising, as it results in the fewest timeouts–5 fewer compared to other values. At the same time, it noticeably improves CPU time for ParallelGC, which makes it competitive with SerialGC, while also retaining much better wall time, even compared to G1GC. Additionally, peak memory consumption is decreased by approximately 31 percent, which means that CPAchecker requires less memory to run, which is particularly beneficial in scenarios where memory is limited.

Table 5: Comparison of garbage collection cycles for SerialGC, ParallelGC, and G1GC across the entire subset of tasks before and after setting the initial and maximum heap sizes equal

| **Configuration** | **Full GCs** | **Young GCs** | **Concurrent marking cycles** |
|---|---|---|---|
| **SerialGC** | | | |
| Default | 15 724 | 330 381 | 0 |
| `Xms=Xmx` | 4 143 | 28 761 | 0 |
| **ParallelGC** | | | |
| Default | 12 189 | 72 202 | 0 |
| `Xms=Xmx` | 3 926 | 33 955 | 0 |
| **G1GC** | | | |
| Default | 166 | 160 278 | 6 997 |
| `Xms=Xmx` | 124 | 102 737 | 5 076 |

Table 6: Results for each garbage collector after tuning the heap size

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **SerialGC** | | | | |
| Default | 2 197 | 148 | 0 | 176 |
| `Xms=Xmx` | 2 197 | 147 | 0 | 174 |
| `MinHeapFreeRatio=20` | 2 194 | 150 | 1 | 180 |
| `MinHeapFreeRatio=70` | 2 198 | 147 | 0 | 175 |
| `MaxHeapFreeRatio=50` | 2 197 | 148 | 0 | 177 |
| **ParallelGC** | | | | |
| Default | 2 193 | 155 | 0 | 183 |
| `Xms=Xmx` | 2 179 | 167 | 0 | 182 |
| `MinHeapFreeRatio=20` | 2 189 | 158 | 1 | 188 |
| `MinHeapFreeRatio=40` | 2 191 | 156 | 0 | 184 |
| `MinHeapFreeRatio=70` | 2 195 | 151 | 0 | 177 |
| `MinHeapFreeRatio=80` | 2 197 | 150 | 0 | 176 |
| `MinHeapFreeRatio=90` | 2 194 | 151 | 0 | 175 |
| `MaxHeapFreeRatio=10` | 2 137 | 219 | 1 | 220 |
| `MaxHeapFreeRatio=50` | 2 186 | 161 | 1 | 189 |
| **G1GC** | | | | |
| Default | 2 198 | 146 | 1 | 184 |
| `Xms=Xmx` | 2 192 | 152 | 1 | 186 |
| `MinHeapFreeRatio=20` | 2 193 | 151 | 1 | 184 |
| `MinHeapFreeRatio=70` | 2 195 | 150 | 0 | 183 |
| `MaxHeapFreeRatio=50` | 2 196 | 148 | 1 | 184 |

Table 7: Performance results of the commonly solved subset for each garbage collector after tuning the heap size

| Configuration | Common Correct Tasks | Avg. CPU time (s) | Avg. Wall time (s) | Avg. Peak memory (MB) |
|---|---|---|---|---|
| **SerialGC** | 2 191 | | | |
| `Default` | | 141 | 116 | 1 730 |
| `Xms=Xmx` | | 138 | 113 | 3 770 |
| `MinHeapFreeRatio=20` | | 145 | 120 | 1 590 |
| `MinHeapFreeRatio=70` | | 139 | 114 | 1 980 |
| `MaxHeapFreeRatio=50` | | 141 | 116 | 1 730 |
| **ParallelGC** | 2 123 | | | |
| `Default` | | 136 | 93.6 | 2 730 |
| `Xms=Xmx` | | 134 | 96.4 | 3 690 |
| `MinHeapFreeRatio=20` | | 140 | 91.8 | 1 420 |
| `MinHeapFreeRatio=40` | | 137 | 91.8 | 1 470 |
| `MinHeapFreeRatio=70` | | 132 | 92.5 | 1 700 |
| `MinHeapFreeRatio=80` | | 131 | 92.6 | 1 870 |
| `MinHeapFreeRatio=90` | | 130 | 93.5 | 1 920 |
| `MaxHeapFreeRatio=10` | | 164 | 91.6 | 1 210 |
| `MaxHeapFreeRatio=50` | | 140 | 91.8 | 1 410 |
| **G1GC** | 2 181 | | | |
| `Default` | | 146 | 102 | 3 000 |
| `Xms=Xmx` | | 147 | 103 | 3 420 |
| `MinHeapFreeRatio=20` | | 146 | 102 | 2 980 |
| `MinHeapFreeRatio=70` | | 146 | 102 | 3 080 |
| `MaxHeapFreeRatio=50` | | 146 | 102 | 2 910 |

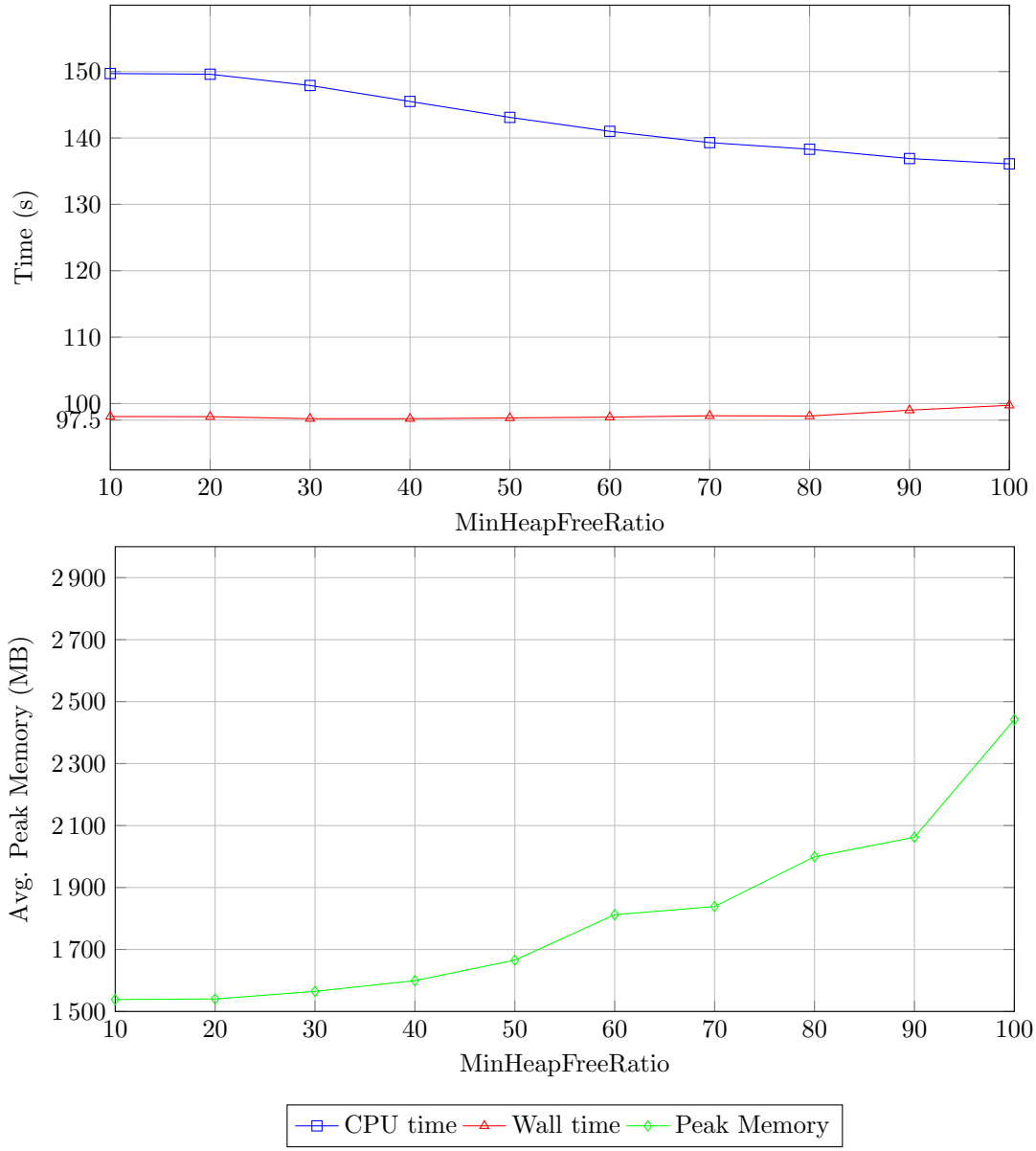
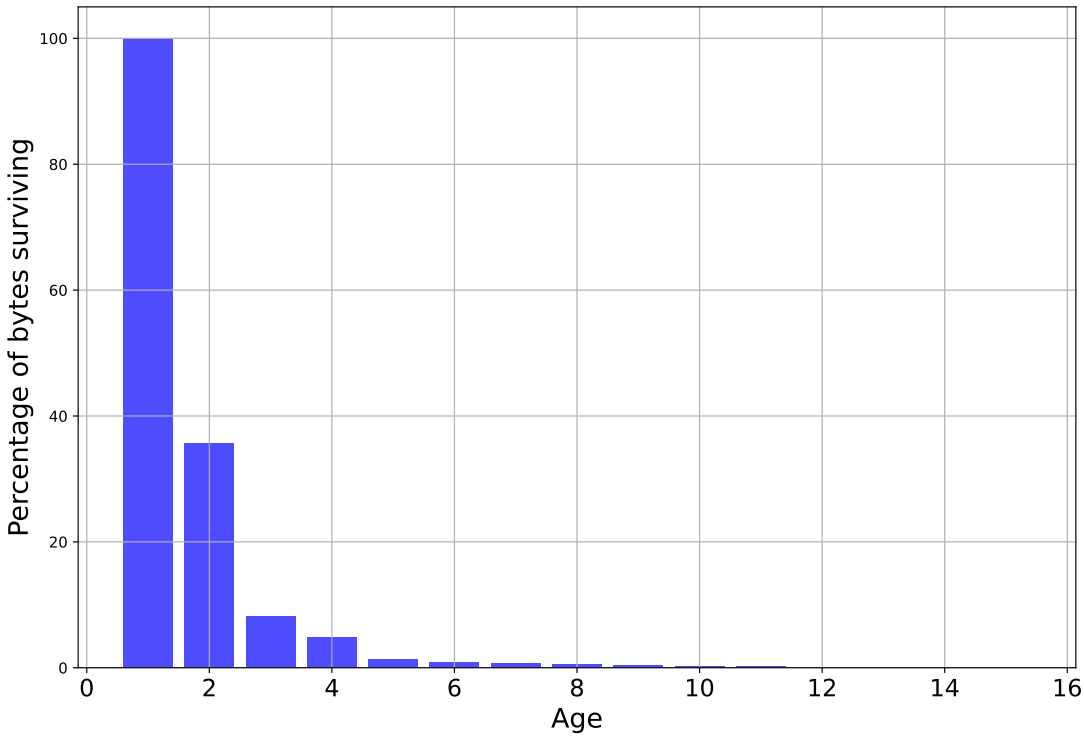



Figure 2: Line plots of performance measures for ParallelGC across various MinHeapFreeRatio settings for the commonly solved subset

### 5.2.4 Generation Size

Before adjusting the generations, we first analyzed the age distribution of objects when executing CPAchecker. As explained in Section 4.2, the age of an object indicates how many GC events it has already survived in the young generation, and the age distribution of objects is measured in bytes. For this analysis, we used SerialGC, as it performs the most frequent young collections, which provides a more accurate dataset than the other GC algorithms. Figure 3 shows the distribution of surviving object bytes by age. Next, we experimented with different sizes of the new and old generations, as well as the survivor spaces, both with and without adaptive resizing of the generations. Table 8 presents a summary of the results and Table 9 shows the average measures for the commonly solved subset of each tuned variant and the default.

Figure 3: Distribution of surviving object bytes by age of all tasks from the subset using SerialGC




It is important to note that age logging begins at age 1, so Figure 3 only provides information on objects that have survived at least one GC event. We observe that of the objects that reach age 1, less than half make it to age 2. Similarly, of those that reach age 2, only about a quarter survive to age 3. Overall, less than 5 percent of objects reach an age of 5 or more. Regarding the tuning of generations, we can see that the different configurations did not lead to any

noticeably improvement in CPAchecker's performance or the number of correct tasks. In some cases, performance even worsened, particularly when adaptive heap sizing was disabled.

We cannot draw conclusions about the overall lifetime distribution of objects based on age plotting alone, as we do not know how many bytes are freed during the first GC event. However, the data does indicate how old the objects get that have survived at least one GC event. In our case, objects tend to have a very short lifespan, with only 5 percent of objects that survive one GC event going on to survive five GC events. One might assume that a larger young generation via `NewRatio=1` could be beneficial, given the large number of short-lived objects. However, this effect cannot be observed, particularly with SerialGC and ParallelGC, where no improvements in CPU time or wall time are achieved, and the number of correct tasks was lower. As described in Section 4.2, a larger young generation may result in less frequent, but more expensive and longer young collections, as more objects need to be processed. In the case of CPAchecker, this trade-off seems to favor a smaller young generation. We may conclude that the default young generation size as well as the adaptive size management are well-suited to this balance, as the age distribution of CPAchecker's objects corresponds with the weak generational hypothesis, which is consistent with the way GC algorithms are implemented [17]. Therefore, this could also explain why changing the generation size has a greater impact when adaptive sizing is disabled.

Overall, we did not find any generation tuning that improved results for our use cases compared to the default settings. Additionally, tuning generations can be challenging to maintain, as the age distribution of objects may vary with different tasks and analyses.

Table 8: Results for each garbage collector after tuning the generations

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **SerialGC** | | | | |
| Default | 2 197 | 148 | 0 | 176 |
| `NewRatio=1` | 2 190 | 156 | 0 | 178 |
| `NewRatio=3` | 2 198 | 147 | 1 | 177 |
| `SurvivorRatio=6` | 2 196 | 150 | 0 | 177 |
| `SurvivorRatio=10` | 2 197 | 147 | 1 | 176 |
| `Xms=Xmx` | 2 197 | 147 | 0 | 174 |
| `Xms=Xmx & NewRatio=1` | 2 190 | 156 | 0 | 175 |
| `Xms=Xmx & NewRatio=3` | 2 192 | 150 | 1 | 176 |
| `Xms=Xmx & SurvivorRatio=6` | 2 199 | 146 | 0 | 174 |
| `Xms=Xmx & SurvivorRatio=10` | 2 184 | 161 | 0 | 177 |
| **ParallelGC** | | | | |
| Default | 2 193 | 155 | 0 | 183 |
| `NewRatio=1` | 2 187 | 163 | 0 | 184 |
| `NewRatio=3` | 2 193 | 155 | 0 | 186 |
| `SurvivorRatio=6` | 2 193 | 153 | 0 | 181 |
| `SurvivorRatio=10` | 2 190 | 156 | 0 | 183 |
| `Xms=Xmx` | 2 179 | 167 | 0 | 182 |
| `Xms=Xmx & NewRatio=1` | 2 166 | 182 | 0 | 184 |
| `Xms=Xmx & NewRatio=3` | 2 188 | 157 | 0 | 180 |
| `Xms=Xmx & SurvivorRatio=6` | 2 178 | 168 | 0 | 183 |
| `Xms=Xmx & SurvivorRatio=10` | 2 185 | 161 | 0 | 180 |
| **G1GC** | | | | |
| Default | 2 198 | 146 | 1 | 184 |
| `NewRatio=1` | 2 198 | 147 | 1 | 182 |
| `NewRatio=3` | 2 198 | 147 | 0 | 184 |

Continued on next page

Table 8: (continued)

| Configuration | Correct Tasks | Timeout | OUT OF MEMORY | Avg. CPU time (s) |
|---|---|---|---|---|
| SurvivorRatio=6 | 2 201 | 142 | 1 | 182 |
| SurvivorRatio=10 | 2 199 | 146 | 0 | 185 |
| Xms=Xmx | 2 192 | 152 | 1 | 186 |
| Xms=Xmx & NewRatio=1 | 2 203 | 142 | 1 | 183 |
| Xms=Xmx & NewRatio=3 | 2 203 | 143 | 0 | 183 |
| Xms=Xmx & SurvivorRatio=6 | 2 196 | 149 | 1 | 182 |
| Xms=Xmx & SurvivorRatio=10 | 2 181 | 164 | 0 | 187 |
| G1NewSizePercent=10 | 2 200 | 143 | 1 | 184 |
| G1NewSizePercent=20 | 2 201 | 144 | 0 | 183 |
| G1NewSizePercent=30 | 2 196 | 149 | 0 | 183 |
| G1MaxNewSizePercent=40 | 2 199 | 145 | 0 | 184 |
| G1MaxNewSizePercent=80 | 2 196 | 148 | 0 | 185 |

Table 9: Performance results of the commonly solved subset for each garbage collector after tuning the generations

| Configuration | Common Correct Tasks | Avg. CPU time (s) | Avg. Wall time (s) | Avg. Peak memory (MB) |
|---|---|---|---|---|
| **SerialGC** | 2 170 | | | |
| `Default` | | 138 | 113 | 1 690 |
| `NewRatio=1` | | 138 | 114 | 2 010 |
| `NewRatio=3` | | 138 | 113 | 1 570 |
| `SurvivorRatio=6` | | 138 | 113 | 1 670 |
| `SurvivorRatio=10` | | 137 | 113 | 1 710 |
| `Xms=Xmx` | | 134 | 110 | 3 740 |
| `Xms=Xmx & NewRatio=1` | | 136 | 112 | 4 620 |
| `Xms=Xmx & NewRatio=3` | | 136 | 111 | 3 230 |
| `Xms=Xmx & SurvivorRatio=6` | | 136 | 111 | 3 620 |
| `Xms=Xmx & SurvivorRatio=10` | | 135 | 111 | 3 800 |
| **ParallelGC** | 2 150 | | | |
| `Default` | | 141 | 96.5 | 2 810 |
| `NewRatio=1` | | 141 | 96.7 | 3 170 |
| `NewRatio=3` | | 143 | 96.9 | 2 550 |
| `SurvivorRatio=6` | | 139 | 96.2 | 2 860 |
| `SurvivorRatio=10` | | 139 | 96.7 | 2 870 |
| `Xms=Xmx` | | 136 | 97.7 | 3 740 |
| `Xms=Xmx & NewRatio=1` | | 137 | 98.9 | 4 590 |
| `Xms=Xmx & NewRatio=3` | | 136 | 97.3 | 3 250 |
| `Xms=Xmx & SurvivorRatio=6` | | 136 | 97.7 | 3 740 |
| `Xms=Xmx & SurvivorRatio=10` | | 135 | 98.9 | 3 910 |
| **G1GC** | 2 170 | | | |
| `Default` | | 143 | 99.6 | 2 970 |

Continued on next page

Table 9: (continued)

| Configuration | Common Correct Tasks | Avg. CPU time (s) | Avg. Wall time (s) | Avg. Peak memory (MB) |
|---|---|---|---|---|
| `NewRatio=1` | | 143 | 99.1 | 2 640 |
| `NewRatio=3` | | 143 | 99.6 | 2 990 |
| `SurvivorRatio=6` | | 143 | 98.9 | 3 180 |
| `SurvivorRatio=10` | | 144 | 101 | 2 840 |
| `Xms=Xmx` | | 145 | 101 | 3 400 |
| `Xms=Xmx & NewRatio=1` | | 144 | 102 | 5 420 |
| `Xms=Xmx & NewRatio=3` | | 144 | 100 | 3 790 |
| `Xms=Xmx & SurvivorRatio=6` | | 144 | 98.9 | 3 370 |
| `Xms=Xmx & SurvivorRatio=10` | | 145 | 102 | 3 430 |
| `G1NewSizePercent=10` | | 143 | 99.7 | 2 840 |
| `G1NewSizePercent=20` | | 144 | 99.2 | 2 720 |
| `G1NewSizePercent=30` | | 143 | 99.1 | 2 670 |
| `G1MaxNewSizePercent=40` | | 144 | 101 | 3 060 |
| `G1MaxNewSizePercent=80` | | 144 | 100 | 2 950 |

### 5.2.5 Number of Parallel and Concurrent Threads

For the evaluation of different numbers of parallel and concurrent threads, we also experimented with configurations using 8 virtual cores. With 8 virtual cores, more variations in the number of threads were possible compared to using 4 virtual cores. However, when evaluating the number of threads, we only compare results with the same number of virtual cores. Table 10 shows the results of these experiments. Figure 4 displays the performance results when changing the number of parallel threads for ParallelGC, while Figure 5 presents the performance results when changing the number of parallel and concurrent threads for G1GC. In both figures, the default values are highlighted in bold. SerialGC is not considered, as it only uses a single thread for GC.

We can observe that for ParallelGC, reducing the number of parallel threads decreases CPU usage, which leads to more correct tasks. However, this comes at the expense of increased wall time and peak memory usage. For G1GC, a

similar trend is noticeable when comparing configurations with the same number of concurrent threads, although the improvement in CPU time is less pronounced than with ParallelGC. When increasing or decreasing the number of concurrent threads compared to the default, there is no improvement in the number of correct tasks or performance measures. Instead, CPU time and wall time increase slightly.

Table 10: Results for ParallelGC after tuning parallel threads and for G1GC after tuning parallel and concurrent threads

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **ParallelGC (8 Cores)** | | | | |
| `ParallelGCThreads=8` | 2 191 | 155 | 0 | 180 |
| `ParallelGCThreads=4` | 2 197 | 148 | 0 | 175 |
| `ParallelGCThreads=2` | 2 200 | 145 | 0 | 174 |
| **ParallelGC (4 Cores)** | | | | |
| `ParallelGCThreads=4` | 2 193 | 155 | 0 | 183 |
| `ParallelGCThreads=2` | 2 197 | 148 | 0 | 179 |
| **G1GC (8 Cores)** | | | | |
| *8 Parallel Threads* | | | | |
| `ConcGCThreads=4` | 2 198 | 148 | 0 | 179 |
| `ConcGCThreads=2` | 2 200 | 143 | 0 | 177 |
| `ConcGCThreads=1` | 2 201 | 142 | 0 | 178 |
| *4 Parallel Threads* | | | | |
| `ConcGCThreads=2` | 2 200 | 143 | 0 | 176 |
| `ConcGCThreads=1` | 2 208 | 135 | 0 | 175 |
| *2 Parallel Threads* | | | | |
| `ConcGCThreads=1` | 2 204 | 140 | 0 | 176 |
| **G1GC (4 Cores)** | | | | |
| *4 Parallel Threads* | | | | |
| `ConcGCThreads=2` | 2 192 | 155 | 0 | 185 |
| `ConcGCThreads=1` | 2 198 | 146 | 1 | 184 |
| *2 Parallel Threads* | | | | |
| `ConcGCThreads=1` | 2 199 | 144 | 1 | 182 |

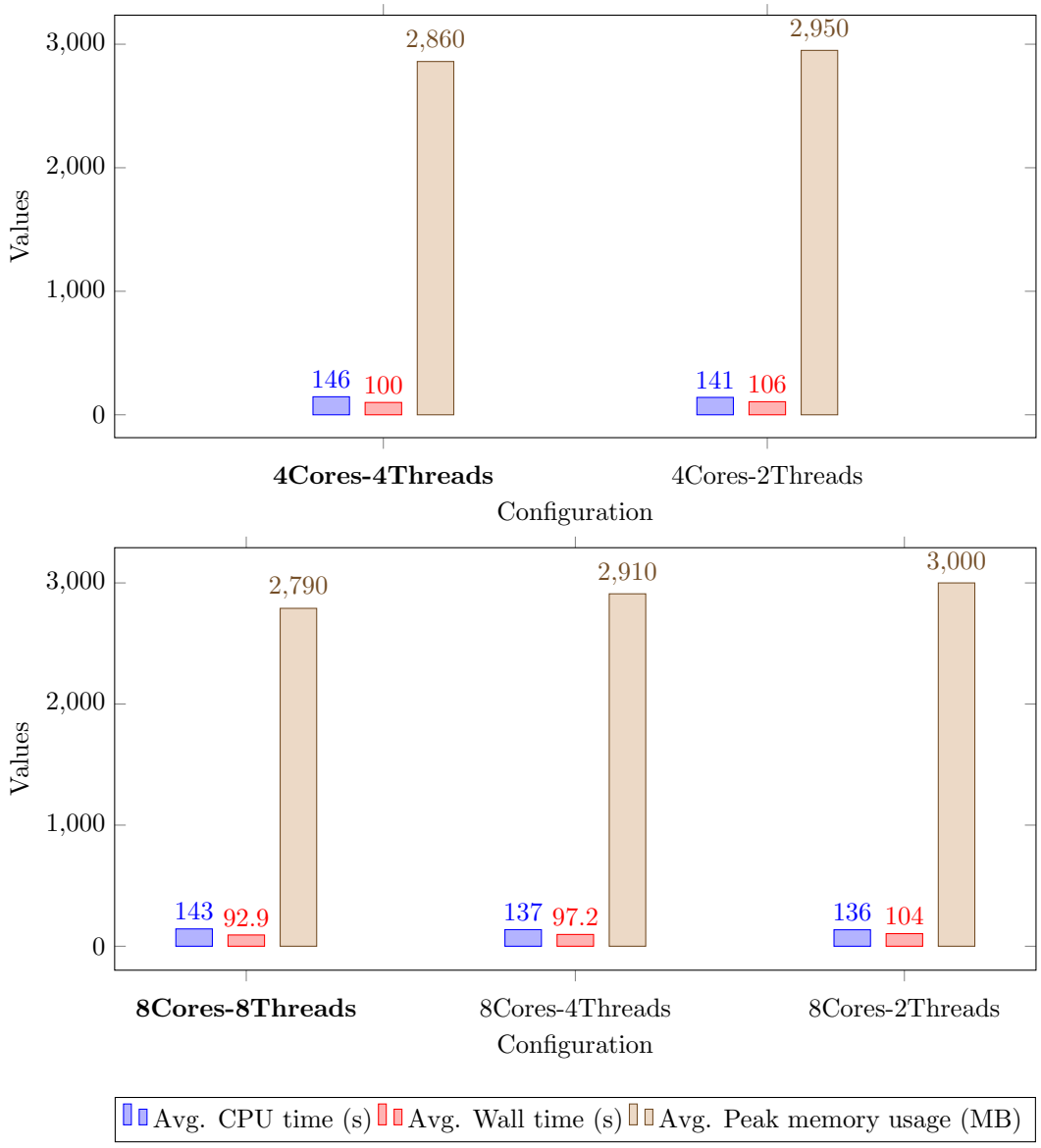



Figure 4: Bar plots of performance results for the commonly solved subset using ParallelGC with 4 and 8 virtual cores, each with varying numbers of parallel threads.

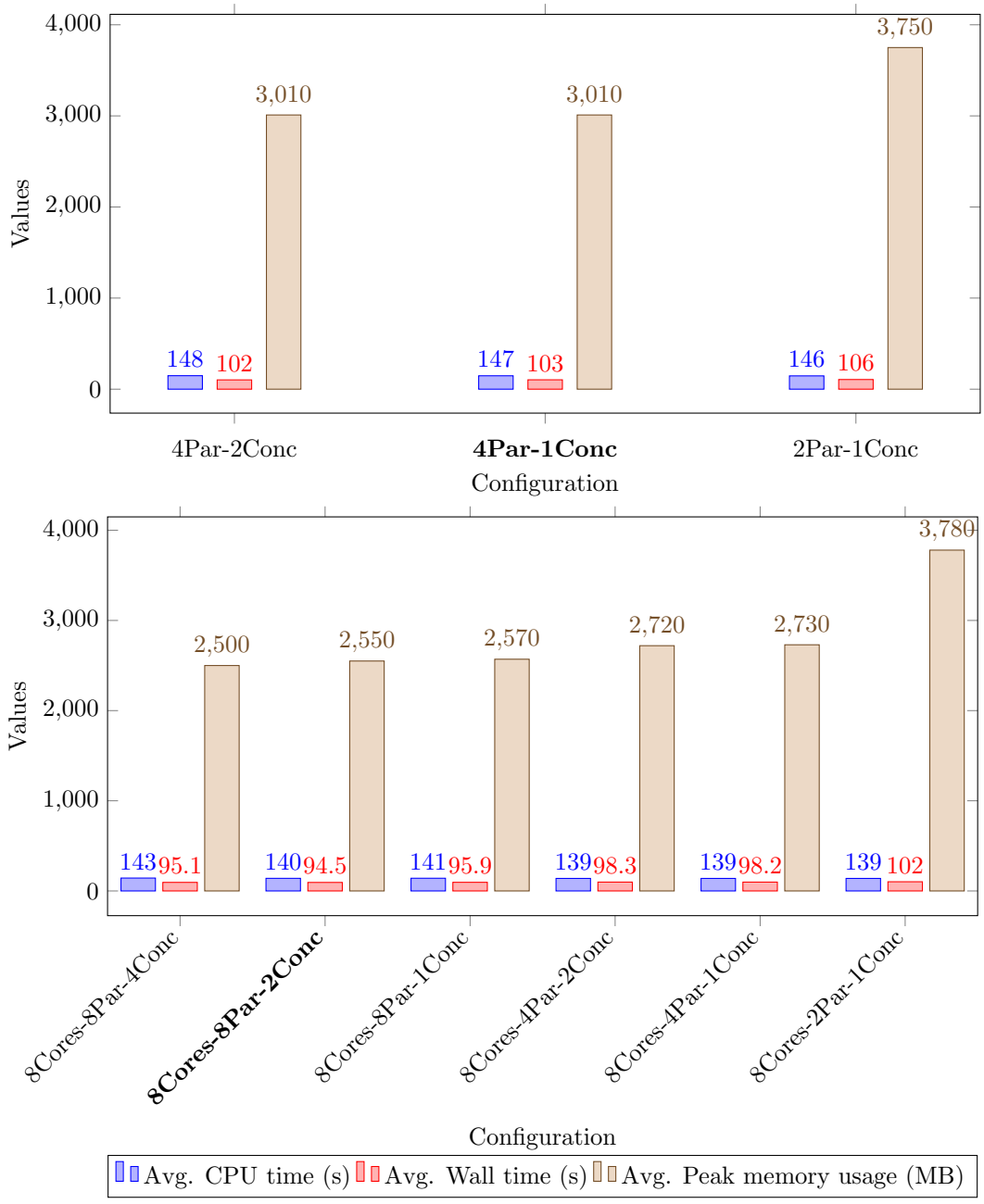



Figure 5: Bar plots of performance results for the commonly solved subset using ParallelGC with 4 and 8 virtual cores, each with varying numbers of parallel (Par) and concurrent (Conc) threads.

Our experiments with CPAchecker confirm the observation by Cai [5], as explained in Section 4.3, that the high level of parallelization in GC by default leads to good wall time at the expense of CPU usage. We may conclude that reducing the number of threads can improve CPU time.

Our considerations in Section 4.3, that fewer threads might result in a lower peak memory consumption due to fewer data structures and reduced fragmentation, are not supported by the results. Instead, we observed a higher peak memory consumption. We assume that fewer threads take longer to complete the same work. Therefore, ParallelGC increases the heap size to shorten pause times and meet its throughput goal, as explained in Section 4.5. The larger heap could also contribute to the improved CPU time. However, following this logic, G1GC should behave in the opposite way to meet its latency constraints and shrink the heap, as discussed in Section 4.6. This would typically reduce peak memory consumption, which was not observed in our results. Nevertheless, the more frequent collections due to the smaller heap might explain why the CPU time improves less with G1GC compared to ParallelGC. These considerations are subject to further research.

Since increasing and decreasing the number of concurrent threads does not improve the results, we may conclude that the default value effectively balances the trade-offs discussed in Section 4.4: More threads could reduce wall time by completing tasks faster, but they also risk throttling mutator threads due to limited CPU resources, while fewer threads might increase wall time by slowing down allocation rate. In terms of CPU time, concurrent threads consume noticeable CPU resources, but too few can lead to more frequent, expensive full collections.

For our use cases, we expect that reducing the number of parallel threads for ParallelGC could be a good way to achieve better CPU time. For G1GC, changing the number of parallel and concurrent threads does not result in major performance differences.

### 5.2.6 Throughput and Maximum Pause Time Goals

We experimented with relaxing the throughput goal for ParallelGC and the maximum pause time limit for G1GC, as these are already set ambitiously by default. Additionally, we benchmarked ParallelGC with a pause time limit, which is not set by default, and G1GC with a higher throughput goal than default.Table 11 shows the results for both garbage collectors. Quantile plots with results for CPU time, wall time, and peak memory consumption for correct tasks are presented in Figure 6 for ParallelGC and in Figure 7 for G1GC.

We can see that for both garbage collectors, differences in CPU time and wall time only become apparent with more difficult tasks. For ParallelGC, relaxing the throughput goal leads to slightly worse average CPU time and fewer correct tasks. However, performance is noticeably worse when the maximum pause time goal is set, resulting in 106 fewer correct tasks. For G1GC, we observe that there are only minor differences, with a tendency for the higher throughput goal to result in slightly higher CPU time and wall time, while the lower maximum pause time goal tends to improve these measures slightly. Peak memory usage is lower for ParallelGC when the throughput goal is relaxed and the pause time goal

is set. However, for the most difficult tasks, setting the maximum pause time goal requires more memory than the default. For G1GC, setting a higher throughput goal than default results in CPAchecker significantly using more memory, while relaxing the maximum pause time goal reduces peak memory usage.

As lowering the throughput goal worsens wall time and CPU time for ParallelGC, this might confirm our assumptions in Section 4.5. When ParallelGC is not required to keep pauses short to achieve the throughput goal, and as it only performs GC during these pauses, the longer pauses result in an overall longer wall time. At the same time, as explained in Section 4.5, the JVM does not need to expand the heap as much, which leads to lower peak memory consumption. However, because the smaller heap needs to be processed more frequently, CPU usage may increase.

For G1GC, allowing a higher maximum pause time improves CPU time and wall time, which supports our reasoning in Section 4.6. With longer pauses allowed, the JVM does not need to restrict the heap size as much to meet pause time goals, which increases peak memory usage. However, this also results in fewer GC events, which appears to be crucial for shorter CPU time and wall time in the case of CPAchecker.

It is plausible that larger impacts on CPU usage and wall time occur for more difficult tasks because the more challenging the task, the more the heuristics may need to intervene to meet the goals, which makes the configuration of these goals even more critical.

We may conclude that the maximum pause time goal has a significant impact on ParallelGC because it is not designed with latency constraints like G1GC. For example, ParallelGC lacks the ability to avoid full collections, which could be particularly important with a smaller heap.

For ParallelGC, the reason for higher peak memory consumption in a few cases when the pause time goal is set remains unclear. For less difficult tasks, the results generally seem to confirm the expected lower peak memory consumption due to a smaller heap.

For our use cases, lowering the throughput goal for ParallelGC and setting a maximum pause time goal are no options, as they result in worse CPU time and wall time. In addition, the maximum pause time goal reduces the number of correct tasks. However, lowering the maximum pause time goal for G1GC is an option, as it improves all performance measures and is relatively easy to maintain.

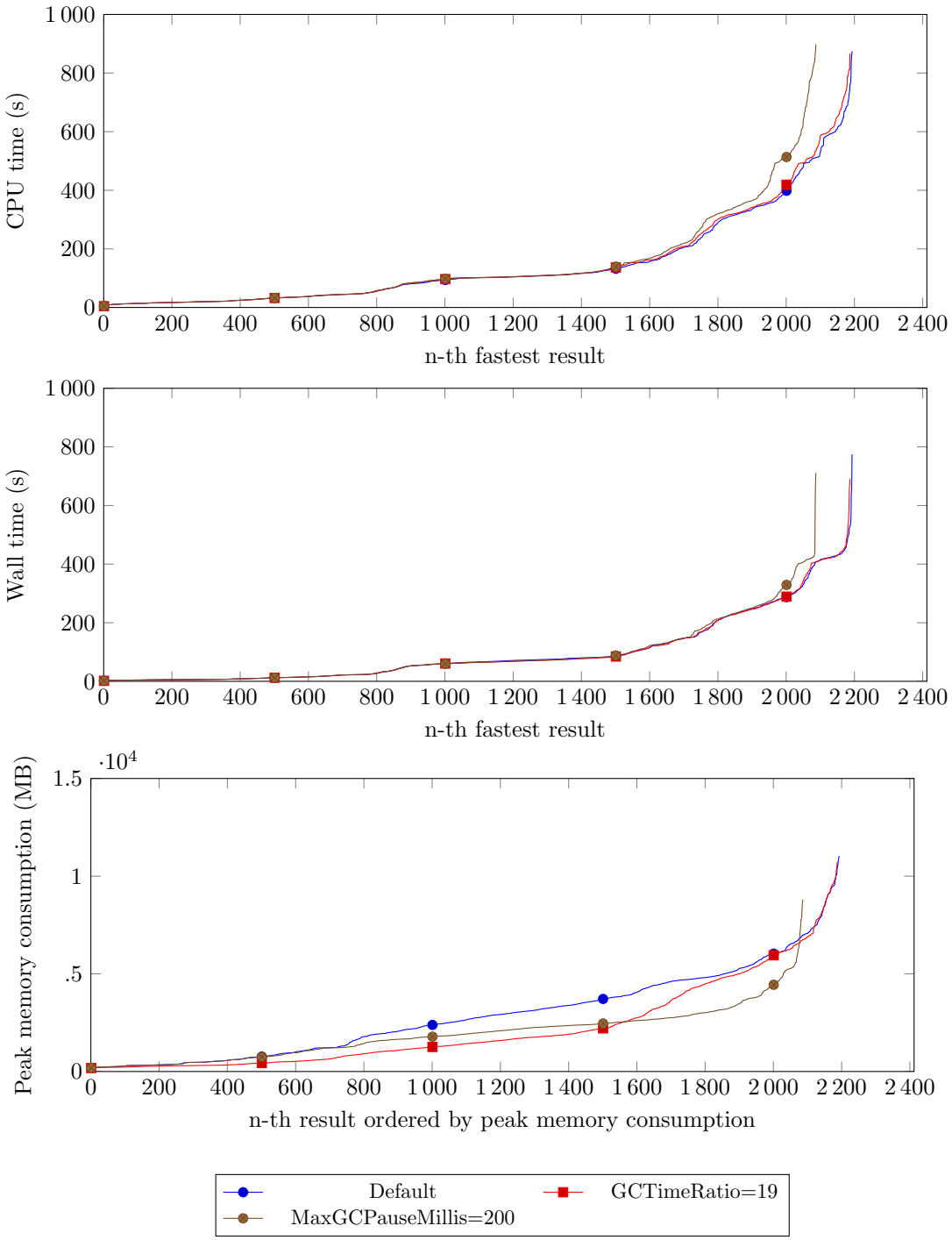



Figure 6: Quantile plots of correct tasks for CPU time, wall time and memory footprint for ParallelGC after tuning throughput and maximum pause time goals

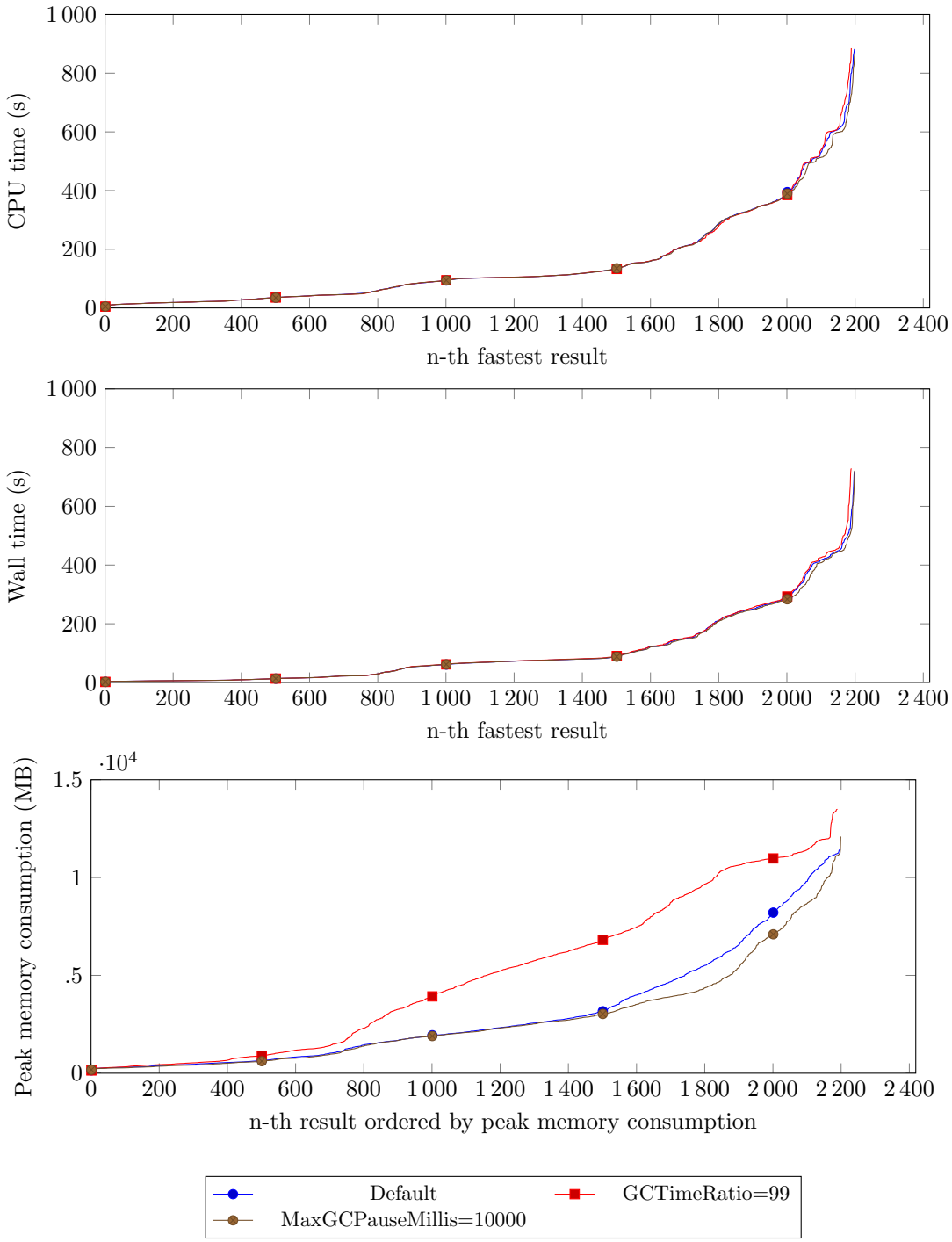



Figure 7: Quantile plots of correct tasks for CPU time, wall time and memory footprint for G1GC after tuning throughput and maximum pause time goals

Table 11: Results for ParallelGC and G1GC after tuning throughput and maximum pause time goals

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **ParallelGC** | | | | |
| Default | 2 193 | 155 | 0 | 183 |
| `GCTimeRatio=19` | 2 187 | 160 | 1 | 188 |
| `MaxGCPauseMillis=200` | 2 087 | 284 | 0 | 212 |
| **G1GC** | | | | |
| Default | 2 198 | 146 | 1 | 184 |
| `GCTimeRatio=99` | 2 189 | 153 | 3 | 184 |
| `MaxGCPauseMillis=10000` | 2 199 | 144 | 1 | 182 |

### 5.2.7 Other Configurations for G1GC

We benchmarked additional configurations for G1GC, none of which led to a significant improvement in CPAchecker's performance. These configurations include:

- `-XX:-G1EagerReclaimHumongousObjects`
- `-XX:+AlwaysPreTouch`
- `-XX:G1HeapRegionSize=16M`
- `-XX:G1HeapWastePercent=10`
- `-XX:G1MixedGCCountTarget=4`
- `-XX:G1MixedGCCountTarget=12`
- `-XX:ReferencesPerThread=0`
- `-XX:ReferencesPerThread=5000`
- `-XX:G1RSetUpdatingPauseTimePercent=60`
- `-XX:+UseLargePages`
- `-XX:+UseStringDeduplication`

- `-XX:-G1UseAdaptiveConcRefinement & -XX:G1ConcRefinementGreenZone=2G & -XX:G1ConcRefinementThreads=0`
- `-XX:-G1UseAdaptiveIHOP & -XX:InitiatingHeapOccupancyPercent=45`
- `-XX:-ReduceInitialCardMarks`
- `-XX:-ParallelRefProcEnabled`
- `-XX:CompressedClassSpaceSize=500M`
- `-XX:G1PeriodicGCInterval=5000`
- `-XX:+UseNUMA`

Details about all flags can be found in the Oracle Tuning Guide [16].

### 5.2.8 Statistical Analysis and Results for the Subset of SV-COMP24

We conducted a statistical analysis on the most interesting GC configurations to validate our findings. This analysis included the three default GC variants as well as three tuning configurations for ParallelGC. Setting `MinHeapFreeRatio=80` for ParallelGC improves CPU time, wall time, and peak memory consumption. Additionally, reducing the number of parallel threads with `ParallelGCThreads=2` offers another option to further improve CPU time, though this could come with trade-offs, as it may increase wall time and peak memory usage. We also included the tuning of ParallelGC with `MinHeapFreeRatio=40`, as explicitly setting this flag–even with the same value as the default–significantly changes the behavior of ParallelGC. Further, we analyzed a tuning configuration for G1GC to assess whether tuning G1GC indeed has a minimal effect. We configured G1GC with the parameters `ParallelGCThreads=2`, `ConcGCThreads=1`, `MaxGCPauseMillis=2000`, and `G1NewSizePercent=20`. We refer to this configuration as "G1GC tuned" throughout the subsequent discussion.

For the analysis, we used a categorical regression model, where the category represents the configuration of a garbage collector. The analysis was performed separately for the three measures CPU time, wall time, and memory peak consumption. We employed the Ordinary Least Squares method to examine the relationship between the dependent variable (each performance measure) and the independent variables (GC configuration). We benchmarked the subset for each configuration five times and calculated the average of the commonly solved subset for each benchmark run. Thus, for each dependent variable CPU time, wall time, and peak memory consumption, we obtained five averages for each configuration.

The model fitting and coefficients can now be interpreted to understand how different GC configurations impact CPAchecker's performance. The model can be expressed as:

$$\text{Measure} =$$
$$\text{Intercept} + b_2 \cdot 1(\text{config} = \text{config2}) + \cdots + b_7 \cdot 1(\text{config} = \text{config7}) + \text{error term}$$

Here, $1(\text{config} = x)$ is an indicator function that equals 1 when configuration $x$ is active and 0 otherwise. The R-squared value defines the proportion of the total variance explained by the model, with a maximum value of 1 indicating a perfect model without any error term. Table 12 shows the results of the statistical analysis. The current default garbage collector of CPAchecker, G1GC, serves as the intercept. The coefficients for the other GC configurations indicate how each performance measure changes on average compared to the intercept. The confidence interval at a 5 percent significance level covers the true parameter value in 95 out of 100 cases. In Figure 8, the number of correct tasks for each GC configuration across the five benchmark runs is presented.

Table 12: Results from categorical regression analysis of CPAchecker's performance under the influence of different GC configurations for the commonly solved subset

| Configuration | Coef | [0.025 | 0.975] |
|---|---|---|---|
| **Avg. CPU time (R-squared = 0.998)** | | | |
| Intercept [G1GC] | 146.8 | 146.6 | 147.0 |
| G1GCtuned | −0.5 | −0.7 | −0.2 |
| *ParallelGC* | | | |
| Default | −0.7 | −0.9 | −0.4 |
| `ParallelGCMinHeapRatio40` | −0.1 | −0.4 | 0.1 |
| `ParallelGCMinHeapRatio80` | −6.8 | −7.1 | −6.6 |
| `ParallelGCMinHeapRatio80 & ParallelGCThreads=2` | −9.4 | −9.6 | −9.2 |
| SerialGC | −5.4 | −5.7 | −5.2 |
| **Avg. Wall time (R-squared = 0.999)** | | | |
| Intercept [G1GC] | 102.4 | 102.2 | 102.5 |
| G1GCtuned | 2.9 | 2.6 | 3.1 |
| *ParallelGC* | | | |
| Default | −2.0 | −2.3 | −1.8 |
| `ParallelGCMinHeapRatio40` | −3.8 | −4.0 | −3.6 |
| `ParallelGCMinHeapRatio80` | −3.3 | −3.5 | −3.0 |
| `ParallelGCMinHeapRatio80 & ParallelGCThreads=2` | 2.2 | 2.0 | 2.4 |
| SerialGC | 14.4 | 14.1 | 14.6 |
| **Avg. Peak Memory Consumption (R-squared = 1)** | | | |
| Intercept [G1GC] | 2 994 | 2 990 | 2 998 |
| G1GCtuned | −19 | −25 | −14 |
| *ParallelGC* | | | |
| Default | −140 | −145 | −134 |
| `ParallelGCMinHeapRatio40` | −1 383 | −1 388 | −1 378 |
| `ParallelGCMinHeapRatio80` | −978 | −983 | −972 |
| `ParallelGCMinHeapRatio80 & ParallelGCThreads=2` | −942 | −948 | −937 |
| SerialGC | −1 258 | −1 263 | −1 252 |

The R-squared value being close to one indicates that our model can explain almost all the variation in the data. This suggests that it accurately reflects the behavior of GC observed in the experiment.

We find that the results of the statistical analysis match with our previous observations. In terms of CPU time, SerialGC in its default configuration gener-

ally performs the best compared to the default variants of ParallelGC and G1GC, with a difference of approximately 3.5 percent on average compared to G1GC. Tuning G1GC results in only minor improvements. However, when tuning ParallelGC with `MinHeapRatio=80`, a slightly better average CPU time can be achieved than with SerialGC. Reducing the number of parallel threads further improves CPU time by up to approximately 6.5 percent compared to G1GC. On the other hand, setting `ParallelGCMinHeapFreeRatio=40` leads to an increase in CPU cycles compared to the default variant of ParallelGC.

When considering wall time, ParallelGC emerges as the fastest, outperforming G1GC, while SerialGC is the slowest by a large margin. Tuning G1GC does not lead to any improvement in wall time. However, tuning ParallelGC with `MinHeapFreeRatio` improves wall time on average, though higher `MinHeapFreeRatio` values slightly increases wall time. ParallelGC with `MinHeapFreeRatio=40` performs the best on average, with a decrease of approximately 3.5 percent in wall time compared to G1GC, while `MinHeapFreeRatio=80` is still approximately 3 percent faster. The tradeoff between performance factors becomes evident here: Reducing the number of parallel threads increases wall time compared to the default ParallelGC, though this effect may be somewhat mitigated by using `MinHeapFreeRatio=80`.

In terms of peak memory consumption, while ParallelGC offers slight improvements over G1GC, SerialGC provides a more substantial reduction. Using SerialGC instead of G1GC, CPAchecker can be executed with approximately 42 percent less memory on average. Tuning G1GC results in only minor reductions in peak memory consumption, whereas tuning ParallelGC with `MinHeapFreeRatio` leads to higher reductions. Notably, setting `MinHeapFreeRatio=40` results in a substantial reduction of approximately 46 percent, while `MinHeapFreeRatio=80` still achieves a reduction of approximately 32 percent. Additionally, lowering the number of parallel GC threads when using `MinHeapFreeRatio=80` does not significantly increase peak memory consumption compared to using `MinHeapFreeRatio=80` alone, while still maintaining a large difference from G1GC.

We can directly derive the variability of CPAchecker's performance from the range of the confidence intervals, as the standard error is directly involved in their calculation. A larger range indicates higher variance in the estimated values. As the largest span is under 7 per mille of the coefficient, the performance of the GC appears to be quite robust against nondeterministic influences, such as small timing differences. This contradicts our initial assumption that the more heuristics are applied, the more variable the performance would be. As small differences in performance are significant, we rounded the results of the statistical analysis to four significant digits.

Regarding the number of correct tasks, we do observe some differences. G1GC achieves the highest number of correct tasks, but the count varies more compared

to other GC configurations. Specifically, SerialGC and the three tuned versions of ParallelGC come close to G1GC in terms of the number of correct tasks, while also showing less variation in the task counts.

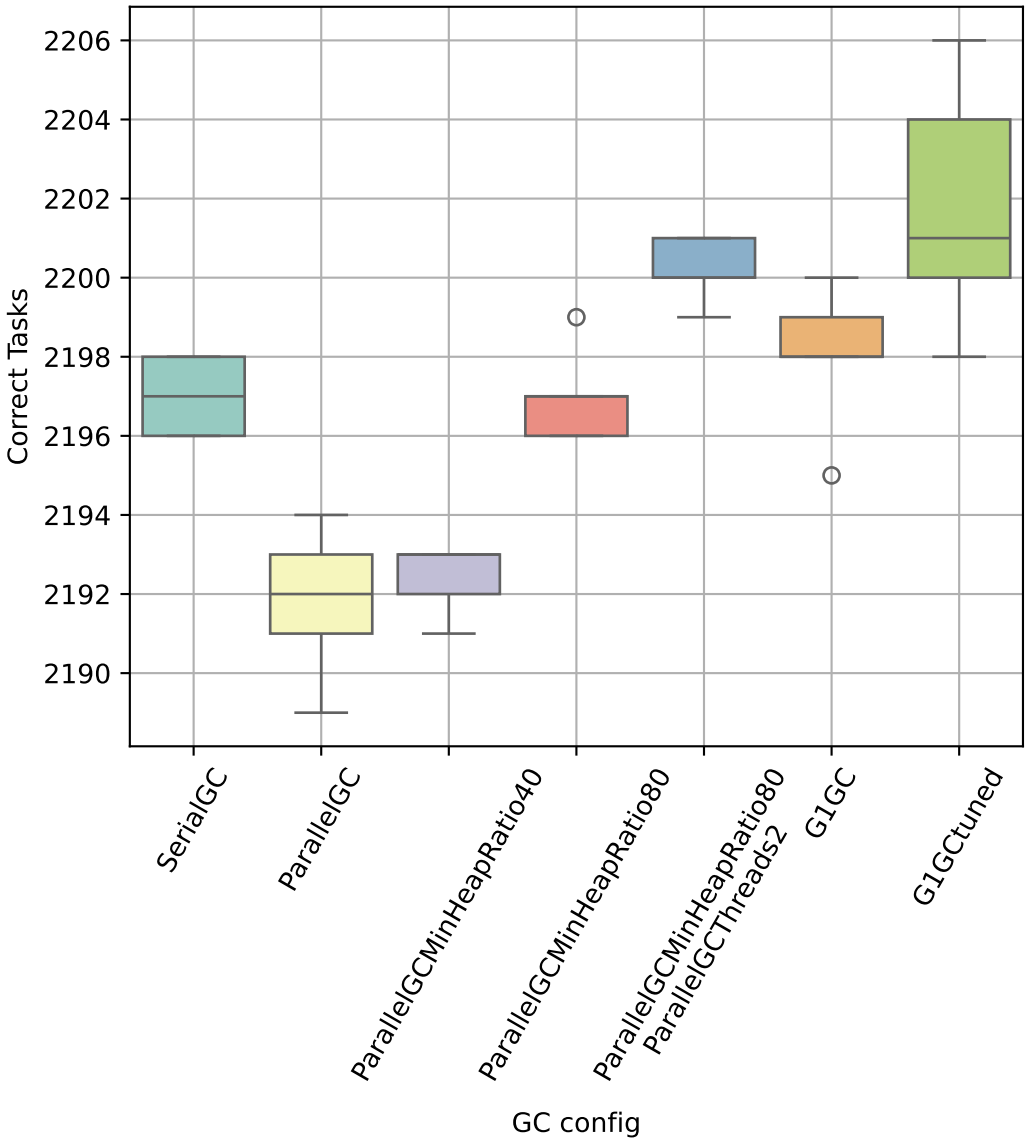



Figure 8: Box plot of correct tasks, with benchmarks executed five times for each GC configuration

For our use cases, tuning G1GC is neither competitive in CPU time nor in wall time or peak memory consumption. Using ParallelGC with `MinHeapFreeRatio=40` is a good option to improve wall time and reducing the peak memory consumption. However, `MinHeapFreeRatio=80` is preferable, as it provides better CPU time while still maintaining good wall time and

low peak memory consumption. In the following, we will therefore no longer consider tuning ParallelGC with `MinHeapFreeRatio=40` or the tuning of G1GC.

### 5.2.9 SV-COMP24

Now we want to examine whether the the results can also scale to the entire set of verification tasks from SV-COMP24. For this purpose, each of the configurations identified in Section 5.2.8 was benchmarked once on the entire number of tasks. The results are presented in Table 13 and the performance results for the commonly solved subset are detailed in Table 14.

Table 13: Results for the whole set of verification tasks from SV-COMP24 for all promising GC configurations

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **SerialGC** | 18 950 | 5 938 | 352 | 219 |
| **ParallelGC** | | | | |
| Default | 18 946 | 5 903 | 391 | 219 |
| `MinHeapFreeRatio=80` | 18 955 | 5 913 | 371 | 218 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | 18 962 | 5 888 | 387 | 217 |
| **G1GC** | 18 961 | 5 802 | 476 | 218 |

Table 14: Performance results of the commonly solved subset of 18909 correct tasks for the whole set of verification tasks from SV-COMP24 for all promising GC configurations

| Configuration | Avg. CPU time (s) | Avg. Wall time (s) | Avg. Peak memory (MB) |
|---|---|---|---|
| **SerialGC** | 66.0 | 54.3 | 524 |
| **ParallelGC** | | | |
| Default | 66.7 | 51.8 | 807 |
| `MinHeapFreeRatio=80` | 65.7 | 51.4 | 585 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | 65.2 | 52.2 | 589 |
| **G1GC** | 67.9 | 52.0 | 779 |

We can see that the effects of the different configurations are the same as with the subset. CPAchecker consumes less CPU time with the tuned variants of ParallelGC than with SerialGC, and with both less than with G1GC. Using ParallelGC with `MinHeapFreeRatio=80` still achieves the best wall time. With SerialGC, CPAchecker has the lowest peak memory consumption, followed closely by the tuned ParallelGC variants. This leads to less "out of memory" errors than with G1GC. We assume that the higher number of "out of memory" errors with G1GC simultaneously reduces the number of timeouts, as the number of correct tasks remains the same. Therefore, G1GC might seem to achieve better CPU time, but this behavior is not desirable. However, quantitatively, the different GC configurations have less impact on CPU time and wall time for the full set than for the subset. We may conclude that this is because the subset was selected based on cases where performance is most strongly influenced by the GC, meaning that configuration changes have a more pronounced effect. This could also be confirmed when considering the average CPU times and wall times. As CPU time and wall time are significantly lower for the complete set of tasks than for the subset, the subset might consist of more difficult tasks. This matches the observation in Section 5.2.1 that the choice of GC configuration becomes more crucial as the tasks become more difficult.

### 5.2.10 Other Analyses

Since SV-COMP24 includes a variety of different analyses, we also experimented with executing CPAchecker specifically for individual analyses. We chose k-induction, predicate analysis, and value analysis for this purpose, as they

are widely applied. We used another set of verification tasks where all tasks where the property is reachability safety were selected, excluding the categories `ReachSafety-Recursive` and `ConcurrencySafety-Main`. We only set a CPU time limit of 15 min per task. Table 15 shows the results, and Table 16 presents the average measures for the commonly solved subset for each analysis.

Table 15: Results for value analysis, predicate analysis, and k-induction for all promising GC configurations

| Configuration | Correct Tasks | Timeout | Out of memory | Avg. CPU time (s) |
|---|---|---|---|---|
| **Value Analysis** | | | | |
| SerialGC | 3 225 | 4 066 | 227 | 294 |
| *ParallelGC* | | | | |
| Default | 3 192 | 4 156 | 227 | 297 |
| `MinHeapFreeRatio=80` | 3 251 | 4 093 | 228 | 294 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | 3 255 | 4 087 | 228 | 293 |
| G1GC | 3 239 | 4 090 | 229 | 295 |
| **Predicate Analysis** | | | | |
| SerialGC | 4 202 | 4 035 | 8 | 287 |
| *ParallelGC* | | | | |
| Default | 4 203 | 4 033 | 8 | 287 |
| `MinHeapFreeRatio=80` | 4 194 | 4 038 | 8 | 287 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | 4 199 | 4 032 | 8 | 287 |
| G1GC | 4 199 | 4 032 | 14 | 287 |
| **k-induction** | | | | |
| SerialGC | 7 398 | 4 604 | 312 | 352 |
| *ParallelGC* | | | | |
| Default | 7 395 | 4 422 | 496 | 346 |
| `MinHeapFreeRatio=80` | 7 403 | 4 443 | 458 | 346 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | 7 407 | 4 430 | 464 | 345 |
| G1GC | 7 403 | 4 452 | 458 | 349 |

Table 16: Performance results of the commonly solved subset for value analysis, predicate analysis, and k-induction for all promising GC configurations

| Configuration | Common Correct Tasks | Avg. CPU time (s) | Avg. Wall time (s) | Avg. Peak memory (MB) |
|---|---|---|---|---|
| **Value Analysis** | 3 183 | | | |
| SerialGC | | 47.0 | 37.4 | 578 |
| ParallelGC | | 47.6 | 30.8 | 963 |
| `Default` | | 47.6 | 30.8 | 963 |
| `MinHeapFreeRatio=80` | | 44.2 | 29.5 | 700 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | | 42.0 | 30.0 | 702 |
| G1GC | | 47.3 | 30.9 | 1 140 |
| **Predicate Analysis** | 4 185 | | | |
| SerialGC | | 51.9 | 42.8 | 402 |
| ParallelGC | | 52.2 | 42.4 | 517 |
| `Default` | | 52.2 | 42.4 | 517 |
| `MinHeapFreeRatio=80` | | 52.0 | 42.3 | 426 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | | 51.5 | 42.2 | 424 |
| G1GC | | 53.5 | 42.6 | 456 |
| **K-induction** | 7 356 | | | |
| SerialGC | | 68.4 | 29.6 | 602 |
| ParallelGC | | 69.4 | 28.8 | 1 130 |
| `Default` | | 69.4 | 28.8 | 1 130 |
| `MinHeapFreeRatio=80` | | 68.8 | 28.6 | 650 |
| `MinHeapFreeRatio=80 & ParallelGCThreads=2` | | 68.0 | 28.6 | 650 |
| G1GC | | 72.0 | 29.4 | 1 060 |

We observe the same effects of the GC configurations as before, with a few exceptions. G1GC consumes less CPU cycles than ParallelGC for value analysis, and SerialGC leads to less CPU usage than ParallelGC with `MinHeapFreeRatio=80` for predicate analysis and k-induction. The number of timeouts and "out of memory" errors with G1GC, unlike in Section 5.2.9, resembled those of the other GC configurations. Interestingly, the quantitative effects of the different configurations vary depending on the analysis. These differences are largest in value analysis, followed by k-induction, and smallest in predicate

analysis. We assume that is because the analyses differ in how much work they offload to native libraries, which operate outside the JVM's heap. The more work an analysis performs outside the JVM, the less influential the GC configuration is for CPAchecker's performance.

### 5.2.11 Use-case specific recommendations

In our specific use case, which aims to improve CPU time in a scientific environment, ParallelGC tuned with `MinHeapFreeRatio=80` and `ParallelGCThreads=2` would be the obvious choice, as it provided the best CPU time on average. However, we recommend using SerialGC, as it delivers similarly good CPU time. In addition, for SerialGC, no tuning is required, which makes it easier to maintain across future JDK versions and different analyses of CPAchecker. Another reason for choosing SerialGC is that SerialGC has the added advantage of lower peak memory consumption, which can reduce the number of "out of memory" errors.

In our other use case, where the focus is on fast results with minimal memory usage, we recommend using ParallelGC with `MinHeapFreeRatio=80`. This configuration provides competitive CPU time and is still faster than SerialGC and G1GC. Moreover, less memory is needed to execute CPAchecker when using the tuned version of ParallelGC instead of the current default garbage collector G1GC. If further improvement in CPU time is desired, the number of parallel threads can be reduced.

These recommendations for CPAchecker are made with the understanding that while the quantitative effects of different configurations may vary across different analyses, an impact of each configuration is consistently present.

# 6 Conclusion

We benchmarked CPAchecker's performance across a variety of GC configurations and used different sets of verification tasks as well as individual analyses within CPAchecker. Our goal was to optimize CPAchecker's performance for our specific use cases: One that prioritizes rapid results while retaining a small peak memory consumption, and another that aims to achieve low CPU time in a scientific environment.

First, we examined the performance of garbage collectors in their default configurations, without any tuning. We found that with SerialGC, CPAchecker required the least memory to execute and the fewest CPU cycles. The best wall time was achieved with ParallelGC. CPAchecker's current default garbage collector, G1GC, could not compete with regard to those measures.

It became clear early on that our performance measures—CPU time, wall time, and peak memory consumption—are connected. For instance, as explained in Section 3.1, ParallelGC achieves its good wall time through a high level of parallelization, which comes at the cost of increased CPU time and peak memory consumption. This connection between measures was further confirmed during the tuning of GC. For instance, reducing the number of parallel threads in Section 5.2.5 led to worse wall time and higher peak memory consumption, but improved CPU time. Not every tuning option has the same effect across all garbage collectors; for instance, setting the initial heap size equal to the total heap size in Section 5.2.3 improved CPU time for SerialGC and ParallelGC but worsened it for G1GC.

There are few tuning parameters that can enhance CPAchecker's performance across all measures. This is achievable when ParallelGC and G1GC deviate from their internal throughput and pause time goals. For instance, configuring ParallelGC with MinHeapFreeRatio prioritizes memory consumption over throughput and leads to an significant increase of CPAchecker's performance in all measures, as detailed in Section 5.2.3. However, lowering the throughput goal itself negatively impacts CPU time and wall time. Lowering the maximum pause time goal improves all three measures for G1GC. Both is explained in Section 5.2.6.

We found that for generation sizing, the default configuration of GC matches well for CPAchecker, as does the number of concurrent threads for G1GC.

Tuning SerialGC and G1GC did not lead to significant better performance

of CPAchecker. For our use cases, no universal GC recommendation can be made. In our specific use case, which aims to optimize CPU time in a scientific environment, we recommend using SerialGC, as it delivers good CPU time, is easy to maintain, and its low peak memory consumption helps prevent "out of memory" errors. For our other use case, we recommend ParallelGC tuned with `MinHeapFreeRatio=80`, as it provides competitive CPU time and is still faster than SerialGC and G1GC, while also having lower peak memory consumption than the current default, G1GC.

We also found that GC performance was generally robust, with CPAchecker's performance showing minimal variability across different GC configurations. However, the quantitative effects of different configurations may vary across different analyses of CPAchecker.

# 7 Future Work

It is a widely accepted view in the literature that tuning GC is highly application-specific [10, 15, 17]. This is indeed true, as factors like the age distribution of objects are crucial for generational tuning. However, we assume that there may be tuning parameters that are largely independent of the application itself and could therefore be applied to other applications as well. Examples include the number of parallel and concurrent threads. Additionally, the throughput and maximum pause time goals may depend less on the application and more on the size of the heap. General assumptions about the effects of these parameters are also mentioned in the literature, though no quantitative studies have been conducted. Therefore, extending these experiments to other applications and different heap sizes could be interesting.

GC depends on the JDK version. As garbage collectors can significantly influence the performance of an application [15], they are constantly being improved. The experiments could be applied to a newer JDK version, with the generational design of ZGC starting from JDK 21 being particularly noteworthy.

In our case, benchmarking CPAchecker with just one virtual core was not practical because it is not intended to run CPAchecker multiple times on a single physical core. When assigning a single virtual core to the benchmarks, one virtual core is utilized, while the second virtual core of the same physical core remains idle. However, it is quite possible that SerialGC would perform even better on a single virtual core, as it was not originally designed for hyperthreading.

In Section 5.2.5 and Section 5.2.6 we have seen that the maximum pause time goal can lead to higher peak memory consumption, especially with more difficult tasks. This contradicts the expected behavior, as we would assume that the heap would be reduced so that fewer objects need to be processed by the garbage collector, which results in shorter pause times. We would like to further investigate the reasons for this behavior.

# References


[1] D. Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*. Ed. by B. Finkbeiner and L. Kovács. Vol. 14572. Lecture Notes in Computer Science. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15. URL: https://doi.org/10.1007/978-3-031-57256-2%5C_15.

[2] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by W. Damm and H. Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 504–518. DOI: 10.1007/978-3-540-73368-3_51. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_51.

[3] D. Beyer and M. E. Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_16.

[4] D. Beyer, S. Löwe, and P. Wendler. "Reliable benchmarking: requirements and solutions". In: *Int. J. Softw. Tools Technol. Transf.* 21.1 (2019), pp. 1–29. DOI: 10.1007/S10009-017-0469-Y. URL: https://doi.org/10.1007/s10009-017-0469-y.

[5] Z. Cai et al. "Distilling the Real Cost of Production Garbage Collectors". In: *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 2022, pp. 46–57. DOI: 10.1109/ISPASS55109.2022.00005. URL: https://doi.org/10.1109/ISPASS55109.2022.00005.

[6] E. W. Dijkstra et al. "On-the-fly garbage collection: an exercise in cooperation". In: *Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, July 23 - August 2, 1975*. Ed. by F. L. Bauer and K. Samelson. Vol. 46. Lecture Notes in Computer Science. Springer, 1975, pp. 43–56. DOI: 10.1007/3-540-07994-7_48.

URL: https://link.springer.com/chapter/10.1007/3-540-07994-7_48.

[7] C. Flood and R. Kennke. *JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector*. Retrieved 2024-07-26. 2014. URL: https://openjdk.org/jeps/189.

[8] C. H. Flood et al. "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. Ed. by W. Binder and P. Tuma. ACM, 2016, 13:1–13:9. DOI: 10.1145/2972206.2972210. URL: https://doi.org/10.1145/2972206.2972210.

[9] R. H. Inc. *shenandoahArguments.cpp*. Retrieved 2024-07-26. URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/share/gc/shenandoah/shenandoahArguments.cpp.

[10] R. E. Jones, A. L. Hosking, and J. E. B. Moss. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press, 2011. ISBN: 978-1-4200-8279-1. URL: http://gchandbook.org/.

[11] S. Karlsson. *JEP 439: Generational ZGC*. Retrieved 2024-07-26. 2021. URL: https://openjdk.org/jeps/439.

[12] P. Lidén and S. Karlsson. *JEP 333: ZGC: A Scalable Low-Latency Garbage Collector*. Retrieved 2024-07-26. 2018. URL: https://openjdk.org/jeps/333.

[13] T. Maget. *Reproduction Package for Bachelor's Thesis 'Evaluation of JVM Garbage Collectors for CPAchecker'*. Retrieved 2024-08-29. 2024. URL: https://zenodo.org/doi/10.5281/zenodo.13468616.

[14] J. Masamitsu. *JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector*. Retrieved 2024-07-26. 2015. URL: https://openjdk.org/jeps/291.

[15] S. Oaks. *Java Performance, 2nd Edition*. O'Reilly, 2020. ISBN: 978-1-4920-5611-9. URL: https://www.oreilly.com/library/view/java-performance-2nd/9781492056102.

[16] Oracle. *Java HotSpot™ Virtual Machine Performance Enhancements*. Retrieved 2024-07-26. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html.

[17] Oracle. *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 11*. Retrieved 2024-07-26. 2024. URL: https://docs.oracle.com/en/java/javase/11/gctuning/hotspot-virtual-%20machine-garbage-collection-tuning-guide.pdf.

[18] T. Schatzl. *JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector*. Retrieved 2024-07-26. 2019. URL: https://openjdk.org/jeps/363.

[19] A. Shipilev. *JEP 318: Epsilon: A No-Op Garbage Collector*. Retrieved 2024-07-26. 2014. URL: https://openjdk.org/jeps/318.

[20] P. Wendler. "Towards Practical Predicate Analysis". PhD thesis. University of Passau, Germany, 2017. URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/509.

[21] M. Williams. *Java Garbage Collection Basics*. Retrieved 2024-07-26. URL: https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html.

# Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe ChatGPT und DeepL verwendet, um Formulierungen für einige Absätze zu verbessern. Diese Formulierungen wurden sodann von mir überprüft und falls nötig, angepasst.

München, den 29.08.2024

Tobias Maget