Heap Size Adjustment with CPU Control

Sanaz Tavakolisomeh
sanazt@ifi.uio.no
University of Oslo
Oslo, Norway

Marina Shimchenko
marina.shimchenko@it.uu.se
Uppsala University
Uppsala, Sweden

Erik Österlund
erik.osterlund@oracle.com
Oracle
Sweden, Stockholm

Rodrigo Bruno
rodrigo.bruno@tecnico.ulisboa.pt
INESC-ID/Técnico, ULisboa
Lisbon, Portugal

Paulo Ferreira
paulofe@ifi.uio.no
University of Oslo
Oslo, Norway

Tobias Wrigstad
tobias.wrigstad@it.uu.se
Uppsala University
Uppsala, Sweden

Abstract

This paper explores automatic heap sizing where developers let the frequency of GC expressed as a target overhead of the application's CPU utilisation, control the size of the heap, as opposed to the other way around. Given enough headroom and spare CPU, a concurrent garbage collector should be able to keep up with the application's allocation rate, and neither the frequency nor duration of GC should impact throughput and latency. Because of the inverse relationship between time spent performing garbage collection and the minimal size of the heap, this enables trading memory for computation and conversely, neutral to an application's performance.

We describe our proposal for automatically adjusting the size of a program's heap based on the CPU overhead of GC. We show how our idea can be relatively easily integrated into ZGC, a concurrent collector in OpenJDK, and study the impact of our approach on memory requirements, throughput, latency, and energy.

CCS Concepts: • **Software and its engineering → Garbage collection**.

Keywords: JVM, Garbage Collection, Heap sizing policy

ACM Reference Format:
Sanaz Tavakolisomeh, Marina Shimchenko, Erik Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad. 2023. Heap Size Adjustment with CPU Control. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617651.3622988

This work is licensed under a Creative Commons Attribution 4.0 International License.

MPLR '23, October 22, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622988

1 Introduction

Garbage collection (GC) offers significant benefits to applications and developers. By abstracting away memory management, code is not tied to a specific strategy for managing heap memory, allowing programs to switch easily between different GC implementations with different properties, e.g., by a command-line argument.

Managed programming languages use various approaches for controlling an application's footprint. Some languages include strategies that automatically reduce the heap size based on memory usage or other metrics. Although the programmer can influence this behavior to some extent by ensuring that objects become garbage, the system may not detect it immediately. Other languages allow the programmer to set an upper bound for the heap size and then manage it relative to that limit. Regardless of how language runtimes manage memory, collecting memory inherently impacts performance in an indirect and hard-to-predict manner.

In OpenJDK HotSpot JVM (OpenJDK for short) a maximum heap size is set on startup, to a user-defined value using the −*Xmx* command-line flag, or in its absence, by picking a default value based on the available memory of the machine (at the time of writing, OpenJDK sets *Xmx* to 25% of the machine's RAM). This decision is made before the program is started, so unless care is taken to explicitly control the maximum heap size, simple programs and complex enterprise applications will share the same memory constraints.

The size of the heap affects the performance differently depending on the GC algorithm. Stop-the-world GC's typically optimize for high throughput and are only able to deliver low latency if the working sets are small enough. In contrast, concurrent collectors typically are not able to achieve as high throughput, but by allowing program activities to continue while GC is running, the frequency or duration of GC has very little impact on a program's performance, as long as the GC is able to collect memory at the same rate as the application is allocating. When memory is abundant or allocation rate is low, infrequent GC can materialize through worse spatial locality in both types of collectors, which may negatively affect performance and/or latency [35].

Table 1. Smallest heap sizes in MB without any allocation stalls for multiple benchmarks across multiple machines. Machines are listed in ascending order based on the number of cores and memory capacity. The lower half of the table displays architectural details for each machine. Machine number 3 is listed three times, indicating configurations with 8, 16, and 24 cores, where the core count was controlled using the `taskset` command. Note that machine #1 could not run Batik without experiencing stalls due to a combination of insufficient memory and inadequate hardware resources for running GC effectively.

	Machines	#1	#2	#3a	#4	#3b	#3c	#5	#6
Heap Sizes	Tomcat	256	512	512	1024	1024	2048	2048	4096
	Spring	4096	4096	4096	8192	4096	16384	4096	32768
	Batik	–	32768	65536	16384	32768	32768	16384	16384
	Luindex	512	512	512	256	512	512	256	256
Hardware Specs	Cores	8	8	8	16	16	24	32	44
	Hardware threads	8	16	16	24	32	48	64	88
	Memory (GB)	16	64	256	128	256	256	16	64
	Core freq (GHz)	2.13	2.3	3.7	3.2/2.4	3.7	3.7	2.7	2.2
	L1 (bytes)	512 KB	64 KB	1 MB	1.5 MB	1 MB	1 MB	64 KB	64 KB
	L3 (bytes)	8 MB	16 MB	128 MB	30 MB	128 MB	128 MB	20 MB	55 MB

A common approach to picking a heap size for an application is by trial-and-error: run the program multiple times with representative load across different JVM instances with varying maximum heap limits and measure its performance until a suitable heap size is identified. A heap size may not be portable across machines and may have to be reevaluated after changes are made to the software or after a switch to a new JVM. This approach may be time-consuming and may not account for variations in the program's memory use during execution. If the maximum heap size is invariant throughout the entire program duration (as in OpenJDK), the entire heap may be used for allocating objects even when memory pressure is low(er), which defers GC and may not be optimal for a program's performance. In conclusion, determining an appropriate heap size for a given application is a complex task that necessitates consideration of various factors. These factors include the hardware configuration of the machine running the program and software-related details such as memory usage patterns.

Automatic heap size adjustment aims to free developers from the need to manually set a heap size, which has proven to be complicated. Instead, developers will be given a sensible default parameter for effective resource management, which should also be intuitive to change. In this work, we explore automatic heap size adjustment in the context of concurrent collectors, where the heap size is controlled by how often we trigger GC, instead of the other way around. As a result, developers can launch Java applications (servers, GUI programs, command line tools, etc.) without having to worry about estimating their memory requirements or worrying that Java's default values might result in these processes ballooning to impractical proportions, thereby disrupting other programs or affecting the application's performance negatively. Our proposal distinguishes itself from previous proposals for automatically adjusting the heap size (e.g., Bruno et al. [6], Grzegorczyk et al. [15], Yang et al. [37], and White et al. [34], cf. §7) by utilising a different "tuning knob" for concurrent collectors. Instead of letting developers control performance through an upper bound on the heap size, we let developers control how much CPU they are willing to spend on GC, expressed as a proportion of the CPU usage of the application. Our strategy is thus more directly tied to performance than heap size, and the heap size becomes a consequence of the GC CPU overhead budget (we call it GC target henceforth). As a result of our choice of tuning knob, the job of picking a reasonable default is easier (or dare we say possible!) than picking a default maximum heap size.

Our contributions can be summarized as:

- **Highlighting heap size variability**: We reveal significant variability of heap sizes across different benchmarks and hardware configurations, emphasizing the impracticality of a one-size-fits-all default heap size. This finding underscores the need for more adaptive approaches. (§2)
- **Exploring automatic heap sizing in a concurrent collector:** We target concurrent collectors whose CPU-intensive activities are not on the critical path of the program's performance. This allows us to dynamically change the heap size to match the program's current behavior and allows developers to trade CPU for memory (and conversely) with minimal impact on performance. (§3)
- **Application in ZGC**: We specifically showcase the implementation and application of our proposed technique on ZGC, a fully concurrent garbage collector. By doing so, we demonstrate its feasibility in a real-world example. (§4)

- **Performance evaluation**: We conduct a comprehensive evaluation demonstrating that adopting our heap size adjustment does not compromise performance or introduce latency issues. Furthermore, we establish that it is possible to determine a sensible default value for CPU overhead that can effectively cater to a variety of applications. (§6)
- **Energy efficiency considerations**: In addition to performance optimization, we illustrate how the concept of CPU overhead can be harnessed as a powerful tool for adjusting the energy spent by an application. (§6)

2 The Perils of Manual Heap Size Picking

When it comes to finding heap sizes, people use multiple rules of thumb, for example, setting heap limits to some multiple of a live set [17]. The challenge becomes even more intricate when considering multiple applications running simultaneously. Kirisame et al. [24] introduced a framework to compare different practices people used for setting up a heap size and derived an optimal "square-root" heap limit rule, which minimizes total memory usage for all applications running together. However, it is still a static heap limit, which might not be optimal on a machine with another architecture, as we demonstrate below.

To study the challenges of manually picking a heap size, we conducted an experiment across multiple machines to find heap sizes for a number of benchmarks. Table 1 shows heap sizes for 4 benchmarks from the DaCapo suite running with the ZGC collector across a range of different machines. Following best practices for tuning heap sizes for concurrent collectors, we tried to find the smallest heap size—expressed as a power of two— that does not produce an allocation stall, a relocation stall, or an OOM (Out of Memory) error for each benchmark on each machine. Ensuring the absence of stalls is of paramount importance when utilizing fully concurrent collectors, given their low-latency nature. Stalls not only lead to performance degradation, as GC becomes critical, but they also undermine the predictability of GC, thereby posing a risk to meeting server-level agreements (SLAs) and latency requirements. Maintaining a consistent and predictable latency profile is essential to uphold performance standards and guarantee uninterrupted service delivery. The reason why we limit ourselves to powers of two is twofold: first, developers have a preference for selecting heap sizes in powers of two [11], and second, finding a stall-free heap size in a reasonable time requires increasing the heap in some increments. In our case, we started at 16MB, doubled the heap size on a stall, and continued our search at the higher heap size. Nevertheless, due to Java's inherited variance, we adopt a stability-oriented approach in which we consider a heap size to be a successful candidate only if three consecutive runs with the same heap size yield no stalls or OOM errors. If stalls or OOM errors do occur, we increment the heap size and repeat the evaluation process.

As it is clear from this experiment, heap sizes vary between the machines without a discernible pattern[1], such as being a function of the number of cores. In addition, we experimented with the same machine, tagged as #3 in the table, with different numbers of cores controlled by taskset: 8, 16, and 24. For Tomcat and Spring, the heap size changes by 4×. So, even within the same machine, modifying core configurations can often require substantial changes in heap sizes. Thus, application deployment across different configurations requires heap sizing to be repeated for each configuration.

3 Heap Size Adjustment with CPU Control

Similar to [34], but in a concurrent setting, we explore an approach where developers directly control memory by setting a *GC target* dictating how much CPU should be spent on GC, expressed as a percentage of the total CPU utilization of the program. Our insight is that memory and GC CPU utilization are inversely correlated. Let's consider a program with a constant allocation rate. When the heap is large, GC occurs infrequently, resulting in low CPU time spent doing GC; conversely, when the heap is small, GC occurs more often, causing a corresponding increase in the CPU time spent doing GC. In a concurrent garbage collector, this kind of trading memory for CPU, or the other way around, should be largely (at least ideally) orthogonal to the program's performance since the program will not block on GC. Furthermore, the program's CPU usage can be considered a proxy for its allocation rate and, by extension, its need for GC. By expressing the GC target in terms of the program's CPU usage, increased program activity immediately translates to increased CPU headroom for GC in absolute numbers. Understanding and controlling the scalability and CPU utilization of a program is a more direct task compared to comprehending its live set, which encompasses all objects contributing to memory pressure.

We define the GC overhead (henceforth denoted GC_{CPU}) as the ratio of time spent doing GC (henceforth T_{GC}) to time spent in the entire application (henceforth T_{APP}):

$$GC_{CPU} = \frac{T_{GC}}{T_{APP}}. \tag{1}$$

These time measurements are the main inputs to our algorithm for determining the new heap size. To mitigate fluctuations, T_{GC} should be calculated using average times for the last n collections (in our implementation, we pick $n = 3$). For instance, if one GC cycle has high CPU activity when the previous cycles did not, it might be too hasty to change the heap size. Thus, the heap size varies "slowly," preventing committing memory that is not needed in the long run.

[1] One anonymous MPLR reviewer suggested that heap sizes might be contingent on both the count of CPU cores and the number of mutators. We sadly currently lack data on the number of mutators, but this aspect presents an intriguing avenue for future exploration.

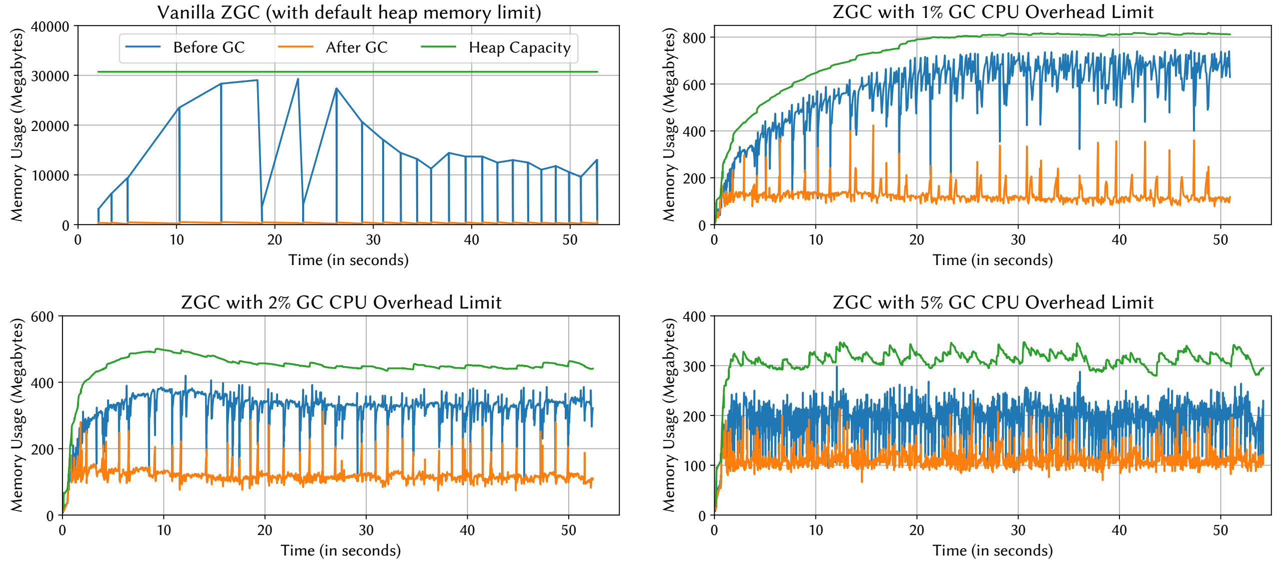

Figure 1. Memory usage of vanilla (unmodified, by default uses 25% of the available RAM) ZGC (22 cycles) and ZGC with 1% (856 cycles), 2% (1506 cycles), and 5% (3182 cycles) GC CPU overhead limits. For this run, we used 12 application threads on a 16-core machine, leaving a 4-core headroom. For each, we measure the following: maximum heap size, memory usage before GC, and memory usage after GC. Note that the y-axis for vanilla ZGC is two orders of magnitude higher. The differences in the x-axes demonstrates the impact of GC on throughput. An artifact of the current ZGC design where each GC cycle forces mutators to take a slow path in the load barrier the first time each reference is loaded. Thus, very frequent GC (i.e., 5%) can materialize as a throughput regression.

The core idea of our proposal is to iteratively adjust the heap size until the GC overhead, i.e., GC_{CPU}, meets the target set by the developer, $Target_GC_{CPU}$. Note that the value is a *target*, not an upper bound. Thus, if $GC_{CPU} > Target_GC_{CPU}$, we increase the heap size to lower the GC frequency and thereby lower the GC CPU overhead. Conversely, when $GC_{CPU} < Target_GC_{CPU}$, we decrease the heap size to trigger more collections, to increase the GC CPU overhead.

To showcase the impact of target GC CPU overhead, we run the Xalan benchmark from the DaCapo suite. It was run four times with vanilla ZGC on machine #3a from Table 1. Additionally, the benchmark was executed with GC targets: 1%, 2%, and 5%. By default, vanilla ZGC uses a high heap memory size (25% of RAM) that is significantly reduced when higher GC target values are used. Figure 1 depicts the results of these runs.

In our approach, during periods of lower CPU activity in the application, a collector will work less, as it is proportional to the application's CPU usage. This results in fewer allocations and overall less pressure on both the allocator and memory manager. Conversely, spikes in the application's activity translate into a higher CPU budget for the GC threads. While it may seem logical to run GC during low CPU activity to utilize available CPU resources, the effectiveness may be limited if there is less memory to free.

At the end of each GC cycle, we compare the GC CPU overhead to the user-defined GC target to calculate $overhead_error_{CPU}$ which we use to adjust the heap size:

$$overhead_error_{CPU} = GC_{CPU} - \frac{Target_GC_{CPU}}{100} \tag{2}$$

We aim to prevent sudden and sharp heap size changes. Therefore, in addition to smoothing out fluctuations in the T_{GC} by considering the average over the last three collections, we avoid using $overhead_error_{CPU}$ directly to modify the heap size, as large error numbers can cause fluctuations in the heap sizes. To mitigate this, we pass the $overhead_error_{CPU}$ through the Sigmoid function [16] to smoothen changes in heap sizes[2]. The Sigmoid function is a mathematical function that is commonly used to model non-linear relationships between variables in statistical models. It maps input values to a range between 0 and 1. Thus, using the Sigmoid function prevents aggressive changes in the heap size. We pass the $overhead_error_{CPU}$ to the Sigmoid function S to calculate

[2] We explored two variations of a step function as well. The first adjusted the heap size proportional to the disparities between CPU overhead and the target. The second involved increasing the soft limit by 50% in either direction if the CPU overhead was above or below the target. Each function led to distinct rates of adaptation and total memory usage. We did not directly compare these three functions against one another; instead, we somewhat arbitrarily opted for the Sigmoid function in our approach.

"Sigmoid overhead error":

$$S(overhead_error) = \frac{1}{1 + e^{-overhead_error_{CPU}}}. \quad (3)$$

We use this result to calculate an *adjustment factor* that limits the changes to the heap size to within a range of 0.5 to 1.5:

$$adjustment_factor = S(overhead_error) + 0.5 \quad (4)$$

When $overhead_error_{CPU}$ is zero, i.e., actual GC CPU overhead equals GC target, the Sigmoid function returns 0.5. Therefore, the *adjustment_factor* becomes 1 and the heap size remains unchanged.

An $S(overhead_error) < 0.5$ means that the actual GC_{CPU} has exceeded the $Target_GC_{CPU}$, so the *adjustment_factor* would be less than 1 and will reduce the heap size, leading to more GC cycles. When the actual GC_{CPU} is below $Target_GC_{CPU}$, $S(overhead_error) > 0.5$, i.e., $adjustment_factor > 1$ will increase the heap size. The heap size will never change more than 50% of the current size (in any direction). Finally, we compute the new heap size as follows:

$$new_size = current_size \times adjustment_factor \quad (5)$$

Our approach can be used in combination with an upper bound on the heap size—e.g., *Xmx*—to trigger an OOM error. However, setting this upper limit may prevent the application from reaching the target GC CPU utilization rate ($Target_GC_{CPU}$). If an upper limit is not specified, the system sets it to a default value, which should be close to the maximum memory available on the machine, but not set to 100% to prevent system instability and swapping. Note, that previously it was 25% of the machine.

4 Prototype Implementation in ZGC

Adjusting the heap size based on GC CPU overhead is suitable for concurrent GCs that do not interfere with the application's critical path. In this section, we implement a prototype on ZGC, a concurrent collector in OpenJDK, to demonstrate the effectiveness of this approach. The prototype follows the ideas presented in the previous section.

4.1 Background on ZGC

The Z Garbage Collector [22] (ZGC) is designed for low latency, offering sub-millisecond pause times invariant of the heap size. GC activity in ZGC occurs concurrently with "mutators" (application threads) by relying on barriers that trap object accesses and coordinate accesses to objects from mutators and GC worker threads. A barrier is essentially some additional logic triggered (in the case of generational ZGC) when a reference is read from a field and placed on the stack or when a reference is loaded from a field. The barrier logic branches on metadata bits embedded in pointers [21]. For example, in the case of a load barrier, if the metadata shows that the pointer is valid, we enter the fast path in which the overhead of the barrier is simply shifting off the metadata bits from the address. Otherwise, we enter the slow path, where we ensure that the pointer is valid by looking up the new canonical address of the object from a forwarding table. This last step may involve copying the object elsewhere and writing to the forwarding table ourselves. ZGC is a multi-phase collector with separate mark and evacuation phases. Its overall design was described by Yang and Wrigstad [36].

Single-generation ZGC uses load barriers to synchronize GC activities with mutators. Generational ZGC [23] instead uses write barriers in addition to load barriers. It maintains a remembered set of references from the old generation to young objects, serving as additional roots during GC in the young generation only. Such a design favors generational workloads where objects are more likely to die young (following the weak generational hypothesis [25]) by supporting a more aggressive collection of the young generation without having to do repeated work on long-living objects. From a resource perspective, generational ZGC requires less CPU and memory usage than single-generation ZGC. In this paper, when we refer to ZGC, we are specifically discussing the generational version of ZGC.

Memory in OpenJDK and ZGC. In addition to *Xmx*, ZGC introduced a new JVM option in OpenJDK 13 called "soft max heap size", and subsequently adopted by G1. A soft max heap size is a limit on the size of the heap, beyond which ZGC *strives* not to grow. Unlike *Xmx*, exceeding the soft max heap size will not result in an OOM error (unless the limit is equal to *Xmx*). When approaching the soft max heap, ZGC triggers GC to bring the heap size below the soft max heap size. If it fails to do so, it will grow the heap instead of going into an allocation stall. The soft max heap size value thus serves as a guiding parameter for GC to balance heap size and allocation rate and has a direct impact on GC activity and frequency. If the value is too small, ZGC might end up doing back-to-back collections. If the value is too large, it can lead to inflated memory costs, floating garbage, heap fragmentation, and poor spatial locality; especially under a low allocation rate.

The relations between different memory parameters in ZGC are shown in Fig. 2. Used memory refers to the occupied memory by both live and dead objects (that have not yet been collected). Maximum capacity or committed memory represents the amount of memory requested by OpenJDK from the Operating System, which is always higher than used memory. In practice, committed memory is often significantly higher: bursts of allocation immediately drive the committed memory up, and to avoid requesting memory from the OS—which may cause delay, or worse, fail—OpenJDK will not return committed memory unless several minutes have elapsed since it was needed (lower bounded by *Xms*, the flag is used to set the minimum and initial heap sizes).

ZGC Heuristics. Heuristics control when to start a GC cycle to avoid running OOM and also how many threads to use for each cycle. In addition, a GC may also be triggered

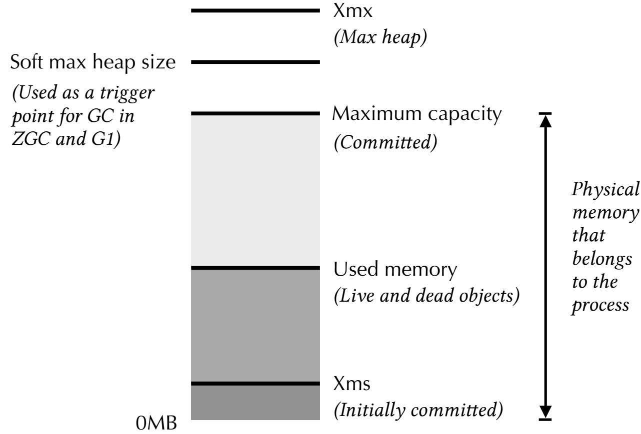

Figure 2. The different heap parameters. *Xmx*: absolute maximum memory for an application. *Xms*: minimum and initial heap. *Maximum capacity*: currently committed memory (above or equal *Xms*, below or equal *Xmx*). *Used memory*: memory occupied by all the objects (above or equal *Xms*, below or equal *Maximum capacity*). *Soft max heap*: point below *Xmx* is used to trigger a GC but will not generate a stall if exceeded, up to *Xmx*.

due to other reasons such as a high allocation rate, high heap usage, or if no collection has been triggered for 5 minutes. Collecting the old generation can also be performed occasionally if not triggered by other reasons. These heuristics consider the available free memory and the time remaining before an OOM error occurs based on the average allocation rate and unforeseen circumstances. To determine the number of GC workers required to prevent OOM, ZGC analyzes the duration of previous GC cycles and adjusts the worker count according to hardware limitations. Finally, ZGC predicts the duration of the next GC cycle based on the number of GC workers and calculates the start time for the next cycle.

4.2 Heap Size Adjustment with CPU Control in ZGC

We take advantage of the aforementioned soft max heap size limit as it has the characteristics we require: it triggers GC but does not stall. To prevent exceeding the machine's heap capacity unintentionally (e.g., due to a too low target), we set *Xmx* to 80% of the available RAM[3] (unless the user has explicitly set *Xmx*). This ensures that the adaptive heap size remains within an upper limit.

Thus, the heap size in our prototype implementation is ZGC's soft max heap, and our technique ultimately results in adjusting the soft heap max up and down at the end of each GC cycle to meet the GC CPU overhead target set by the programmer. The amount of memory committed from the OS by OpenJDK is limited (as usual) by *Xmx* and will only grow in tandem with the soft max heap.

[3]This number reflects a pragmatic choice motivated by wanting to keep some spare memory for remaining programs running on the machine and also to leave space for ZGC's forwarding tables which are allocated off-heap and may grow very large under certain circumstances [26].

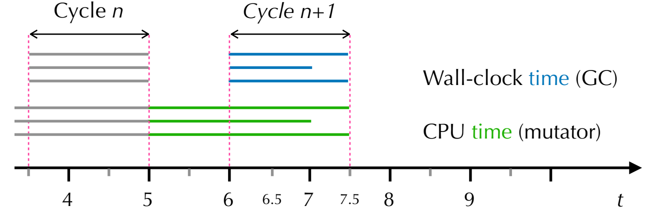

Figure 3. Concrete measurements of GC and application time in our implementation. At the end of the GC cycle $n+1$ ($t = 7.5$), we consider the time spent in GC threads (blue) and the time spent in mutators (green). Gray lines denote time measured at the end of GC cycle n. We only include the time when mutators were *scheduled*, meaning $C_{APP} = 2.5+2+2.5 = 7$. In the case of W_{GC}, we measure from the start to the finish of the GC cycle. Thus, $W_{GC} = 3 \times 1.5 = 4.5$, even though the 2nd GC thread was not scheduled after $t = 7$. Thus, $GC_{CPU} = \frac{4.5}{7} \approx 64\%$. (This example omits barriers, read more about them in §4.3)

4.3 Obtaining T_{GC}

We calculate T_{GC} as the sum of time spent on young (W_{young}) and old collections (W_{old}) plus an estimate of the time mutators spent in the slow path of barriers (B). For simplicity and to avoid adding logic contributing to GC overhead, we use existing telemetry in ZGC. Thus, W_{young} and W_{old} are wall-clock time measurements. For traceability, we prefix wall-clock time measurements by W and CPU time measurements by C below. Thus, we will henceforth write W_{GC} instead of T_{GC} to highlight that the time measurement is a wall-clock time. To address potential inaccuracies in individual measurements, we calculate W_{young} and W_{old} using the average times for the last 3 collections (as we described in §3). For uniformity, we use a single formula (Eq. (6)) to describe W_{GC} and, in a minor collection, set W_{old} to 0.

$$W_{GC} = W_{young} + W_{old} + B \tag{6}$$

As already mentioned, B is the mutator time spent in the slow paths of barriers. When mutators hit slow paths in barriers, they do GC work, either remapping an old address to a forwarding address or performing relocation. We measure the wall-clock time of barriers using sampling: we record the time once for every 1024 slow paths taken, calculate the average time spent in slow paths, and multiply that with the number of slow paths taken.

ZGC calculates GC time separately for each generation by adding the times for the serial and parallel work. The serial time is the wall-clock time spent on non-parallel tasks like relocation set selection after marking, while the parallel time is the sum of the wall-clock time spent by worker threads

on parallelizable tasks.

$$W_{young} = W_{serial_young} + W_{parallel_young} \quad (7)$$

Similarly, for activity in the old generation:

$$W_{old} = W_{serial_old} + W_{parallel_old} \quad (8)$$

4.4 Obtaining T_{APP}

The application's average time is the sum of the scheduled time of all threads spawned by the process (i.e., a CPU time measurement) between two collections in the same generation. Thus, we write C_{APP} henceforth to clarify the nature of T_{APP} in our implementation:

$$C_{APP} = C_{GC_i} - C_{GC_{i+1}} \quad (9)$$

Similarly to GC time, we reuse existing GC telemetry to capture application time to avoid additional measurement overheads. Application time is obtained by measuring CPU time (see Figure 3 for an overview). Listing 1 shows the code for measuring the CPU time of the process.

Listing 1. Code that calculates the process CPU time at moment of the call.

```
1  double ZAdaptiveHeap::process_cpu_time() {
2    timespec tp;
3    int status =
4      clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tp);
5    if (status != 0) {
6      return -1.0;
7    } else {
8      return double(tp.tv_sec) +
9             double(tp.tv_nsec) / NANOSECS_PER_SEC;
10   }
11 }
```

The function `clock_gettime` measures the CPU time consumed by a process, meaning that it includes the CPU time consumed by all threads in the process, including application threads, GC threads, compiler threads, etc. To measure the CPU time between two moments in time, we cache the last result and subtract it from the result of the subsequent call.

4.5 Calculating a Suggested Heap size

Most of our modifications to ZGC are located in its heap sizing mechanism: class `ZAdaptiveHeap`. The main logic is captured in the method `ZAdaptiveHeap::adapt` (see listing 2), which performs the calculations outlined in Sections 3 to 4. For clarity, we add comments with the labels from the equations to aid in mapping the C++ code to the descriptions above. The method is called at the end of each GC activity in both major (young + old) and minor (only young) collections.

Listing 2. The modified `adapt` method that recalculates heap limits in ZGC. (For simplicity, we only show the logic for major GC and remove one lock to reduce clutter.)

```
1  void ZAdaptiveHeap::adapt(ZGenerationId generation,
2   ZStatCycleStats stats) {
3    ZGenerationData& generation_data =
4      _generation_data[(int)generation]; // holds the historical
5
6    double time_last = generation_data._last_cpu_time;
7    double time_now = process_cpu_time(); // see listing 1
8    generation_data._last_cpu_time = time_now;
9
10   // calculate C_APP as in (9)
11   double total_time = time_now - time_last;
12   // record C_APP to calculate averages
13   generation_data._process_cpu_time.add(total_time);
14
15   // Obtain the number of barriers triggered
16   size_t barriers =
17     Atomic::xchg(&_barrier_slow_paths, (size_t)0u);
18   // Obtain average barrier time
19   double barrier_slow_path_time=_barrier_cpu_time.davg();
20   // Calculate B in (6)
21   double avg_barrier_time =
22     barriers * barrier_slow_path_time;
23   double avg_gc_time = stats. // (7) or (8)
24     _avg_serial_time + stats._avg_parallelizable_time;
25   // recalculate C_APP using historical data to smoothen the curve
26   double avg_total_time =
27     generation_data._process_cpu_time.davg();
28
29   double avg_generation_cpu_overhead =
30     (avg_gc_time + avg_barrier_time) / avg_total_time;
31   Atomic::store(&generation_data._generation_cpu_overhead
32     , avg_generation_cpu_overhead);
33
34   double young_cpu_overhead =
35     Atomic::load(&young_data()._generation_cpu_overhead);
36   double old_cpu_overhead =
37     Atomic::load(&old_data()._generation_cpu_overhead);
38   double cpu_overhead = // Calculate W_GC as in (6)
39     young_cpu_overhead + old_cpu_ov or noterhead;
40   double cpu_overhead_error =
41     cpu_overhead - (ZCPUOverheadPercent / 100.0); // (2)
42   double cpu_overhead_sigmoid_error =
43     sigmoid_function(cpu_overhead_error); // (3)
44   double correction_factor =
45     cpu_overhead_sigmoid_error + 0.5; // (4)
46
47   if (is_enabled()){
48     // Call into ZGC to resize the heap, c.f. §4.6
49     ZHeap::heap()->resize_heap(correction_factor);
50   }
51 }
```

4.6 Bounding the Heap Size

After calculating the new suggested heap size, the listing below depicts how we ensure that the result of the calculation falls within a given range. If it does, the function returns `suggested_heap_size`. If it is less than `lower_bound`, then the function returns `lower_bound`. Also, if it is greater than `upper_bound`, the function returns `upper_bound`.

```
const size_t upper_bound = // select smallest of two
  MIN2(soft_max_capacity, current_max_capacity);
const size_t lower_bound = // select smallest of two
  MIN2(1.1 * used(), upper_bound); // 10% extra headroom

const size_t selected_capacity =
  clamp(suggested_capacity, lower_bound, upper_bound);
```

We establish a lower bound for the suggested heap by using the amount of *used* memory. This is because we aim to avoid triggering GC more often than necessary. If we set the suggested heap size below the *used* value, we risk triggering GC when there are no objects to clean. Although concurrent GC does not interfere with the application's critical path (as it is running concurrently and does not force the application to stop) and therefore it might have a negligible impact on performance or latency, the additional GC work can have a negative impact on energy consumption.

4.7 Initial and Adapted Heap Sizes

We set the initial heap size to 16 MB, in terms of the soft limit. This is an unlikely heap size for most programs and will trigger GC as the limit is approached or exceeded which will cause GC to adapt the heap size and (most likely) increase the soft limit (by at most 50% each time). Fig. 1 shows the frequent increases of the soft limit in green. (The top-left sub-figure shows Vanilla ZGC where the soft limit is equal to *Xmx* and never exceeded.) If the soft limit is not exceeded, and the GC overhead is below the target, we will decrease the soft limit to trigger GC more often. This is clearly visible in the two bottom subfigures of Fig. 1.

5 Evaluation

In this section, we are going to answer the following question: How effective is our automated heap sizing strategy, based on CPU usage as a tuning knob, compared to vanilla ZGC which relies on setting a maximum heap size? We now explain our experimental setup and benchmarking methodology.

5.1 Hardware and Software

We evaluate our work by comparing our modified ZGC with its unmodified base also referred to as vanilla (generational ZGC in OpenJDK version 21). We used an Intel Xeon *Sandy-Bridge* EN/EP server machine (machine #5 in Table 1) running Oracle Linux Server 8.4. The machine has 32 identical CPUs, which we configured as a single NUMA-node to avoid NUMA effects. The CPU model is Intel® Xeon® CPU E5-2680 with 64KB L1 cache, 256KB L2 cache, a shared 20MB L3 cache, and 30GB RAM. The configuration allows us to obtain energy consumption statistics.

5.2 Benchmarks

We use the DaCapo benchmark suite (Chopin branch), which includes a variety of microbenchmarks and real-world applications that stress the JVM and the garbage collector. The suite includes several latency-sensitive applications that require low-latency response times. These benchmarks measure metered latency, including request serving time, queuing delays, and interruptions like GC. By using these benchmarks, GC performance can be evaluated in terms of both throughput and responsiveness. We excluded the benchmarks Kafka and JME due to a low CPU utilization issue, as well as Lusearch due to a high CPU utilization variability, making it hard to draw any meaningful conclusions (noted by the benchmark maintainers). We also excluded H2 due to a reproducible memory leak across multiple machines and garbage collectors. When referring to DaCapo in this paper, we specifically mean the DaCapo Chopin benchmark suite. We included all throughput-oriented benchmarks except Cassandra, which is incompatible since OpenJDK 16.

In order to obtain a more comprehensive understanding of our prototype, we also include the Hazelcast benchmark [13]. Hazelcast was chosen since most of the latency-sensitive workloads in DaCapo were excluded for the aforementioned reasons. As low latency is the main goal of a concurrent collector, we wanted to study more such workloads. Hazelcast is designed to provide distributed and scalable in-memory data storage and processing, which can help reduce data access and processing latency.

DaCapo. We use a commit (number 300acaa7) that includes latency-oriented benchmarks as evaluating latency is crucial for fully concurrent collectors. We conducted the benchmarks using the *large* size for all applicable tests. For the remaining benchmarks (Fop, Zxing, Xalan), we used the *default* size.

Hazelcast. Hazelcast performs real-time stream processing. We used all the suggested configuration parameters [31]. It has a fixed workload, set by its key-set size. We experiment with multiple key-set sizes: 400 000, 250 000, 100 000. Thus, we report the results of those 3 different configurations.

5.3 Benchmarking Methodology

We run each benchmark using 5 JVM instances, which lets us identify performance anomalies and outliers that might not have been discernible using a single JVM instance. Note that the variation between JVM instances is within the variation between the last 5 stable iterations of a single JVM. Inside each JVM instance, each benchmark repeats multiple iterations (varies across benchmarks to reach a coefficient of variation (CV) for the last 5 iterations below 5% with respect to execution time), which is necessary to avoid impact from warmup and JIT compilation. Notably, our approach takes

time to adjust from the initial heap size before stabilizing around a GC target.

Once we reached a steady state, we calculated the arithmetic mean of the last 5 iterations to remove noise from the environment. However, in cases where a steady state could not be reached, we used all recorded values for the last 5 iterations per JVM instead of computing the arithmetic mean. This is because taking a mean could hide outliers, and we do not know the shape of the data distribution.

In summary, we compute either one arithmetic mean per JVM instance (resulting in 5 data points in the final set) or all values from each JVM (resulting in 25 data points in the final set). We use the same approach for both adaptive ZGC and vanilla ZGC and to compare them, we perform statistical analysis on the final data sets. The final results reported in Table 2 were calculated using an arithmetic mean of the final set.

5.4 Statistical Analysis

We used different tests to verify the validity and reliability of the results. We perform statistical analysis on the final data sets to draw our conclusions. We employed Welch's t-test [33], Grubb's outlier test [14], and Yuen's t-test [38] to determine whether the differences between the means of the compared results from vanilla and adaptive ZGC are statistically significant. Welch's t-test and Yuen's t-test are particularly useful in cases where we can not make assumptions about the shape of data distribution and the variances of the compared groups are not equal. We believe these tests are safer to use instead of relying on a non-verifiable assumption about the normality of our data distribution.

We used Grubb's outlier test to check if the data set has statistical outliers. If so, we use Yuen's t-test instead of Welch's t-test. Yuen's test involves trimming a fixed proportion of the extreme values from each data set, we used 10%, to reduce the influence of outliers. To determine whether the results exhibit significant differences, we used the p-value obtained from Welch's t-test (with a significance level of 0.05). If the resulting p-value is greater than 0.05, we conclude that the data sets do not exhibit significant differences.

To help provide an overview, we color code the results if statistical significance was found. Red means the adaptive approach is worse than the vanilla ZGC; green means the opposite. White indicates the results are statistically the same. We also highlight a bigger than 5 % negative impact of our approach with a darker shade of red (Table 2, Table 3).

5.5 Energy Measurements

Energy consumption was measured using the Running Average Power Limit (RAPL) [19] interface available on recent Intel architectures. This interface allows machine-specific registers (MSRs) to be read, which contain energy scores. To calculate the final energy score, we report the sum of the package and DRAM domains, following the method used by Shimchenko et al. [30]. Our approach for measuring energy consumption is similar to that used for measuring throughput and latency. For DaCapo benchmarks, warmup iterations were excluded, and statistics were aggregated across 5 JVM instances for the last 5 iterations in each run. For the Hazelcast benchmark, we report energy consumption for the entire run, as it is a longer-running benchmark where the warmup period is a small fraction of the total run time.

5.6 Baseline Heap Sizes

If the *Xmx* option is not specified by the user when starting the JVM, the JVM will default the maximum heap size to 25% of the physical memory available on the system.[4] Research papers that involve measurements across multiple garbage collectors use other collectors like G1 [7] or Serial GC to pick the minimum heap size [29] and then employ a scaling factor to provide additional headroom (additional memory space) for other collectors. However, it is not at all clear if such an approach reflects the actual heap sizes chosen by developers for production systems. For example, developers often tend to choose heap sizes that are powers of two [11].

Proper configuration of a concurrent collector should avoid allocation stalls as these introduce jitter and hurt latency. Thus, we decided to adopt a manual heap size adjustment strategy for our baseline (vanilla ZGC), where we pick the smallest power-of-two heap size with which the application runs reliably without stalling. We use this value for each benchmark as *Xmx* for a baseline configuration in vanilla ZGC; also, we explicitly set *Xms* to 16MB, which is the same as its default value according to the ZGC codebase. Finally, we had to manually pick heap sizes as there is no "best option". We pick baseline values not in order to "beat" something but explain the behaviour of our system.

5.7 GC Targets

To investigate the implications of our proposed design, we studied the impact of GC targets on latency and throughput using varying percentages of GC CPU target overhead. Specifically, we examined the following GC targets: 5% (to a limited extent using 3 JVM instances To assess if the picked list of GC targets is representative, we found the actual GC CPU Overheads without the heap size adjustment for memories picked according to Table 2. Looking at Table 2's GC CPU overheads, the actual GC CPU overheads have a big variation from less than 1 % (Sunflow) to 23 % (Fop). Given that having a closer GC target to the actual GC overhead might better reveal the effect of our adaptive solution, we only evaluated our strategy with a 5% GC target for the benchmarks with actual GC overheads below 5%. This required running additional iterations to reach a steady state, adding time to benchmarking.

[4] https://docs.oracle.com/en/java/javase/19/gctuning/ergonomics.html

Methodologically, testing very small GC targets on short-running benchmarks is challenging since it takes time to grow the heap from the initial 16MB to a size that sustains the required GC target. If this time exceeds the benchmark's run-time, it never reaches a steady state, causing the results inconclusive. Very high GC targets do not represent a real deployment. This said we believe that a picked range of GC targets is sufficient to demonstrate how our system behaves and showcase main trends.

6 Results

We now compare the performance of running the vanilla ZGC with manually selected heap sizes against our adaptive technique, which leverages different values of GC CPU overhead. Throughout the experiments, we closely examined various metrics, including memory usage, execution time, latency, and energy consumption. We sought to identify the advantages and drawbacks of each approach. Additionally, we propose an optimal default value for the GC CPU overhead that strikes a balance between efficient resource utilization and overall system performance.

Prior to presenting our results, we would like to address the absence of 3 benchmarks, Batik, Jython, and Pmd, from our study. These benchmarks have actual GC targets of 80 %, 76 %, and 170 %, respectively, using the maximum memory available on the SandyBridge machine. Therefore, we were unable to allocate additional memory to lower the GC targets for these benchmarks. Nevertheless, our methodology remains valid, and we were able to obtain results for these benchmarks by running them with the maximum available memory on the machine. As a result, the GC CPU overhead of these benchmarks remained similar to their actual values. Note that failing to attain a CPU target does not result in the failure of benchmark execution. The observed outcome is merely a disparity in real CPU overhead when compared to the requested target. If the target is set lower than the actual value and insufficient memory is available to elevate it, the application will persist in running without reaching the target. This situation remains unchanged unless the entire machine's memory suffices to prevent OOM issues, a scenario shared by Vanilla ZGC. Conversely, when the target surpasses the real CPU overhead and reducing memory fails to rectify it, this signifies an absence of substantial GC work. Irrespective of these scenarios, the application continues to function without interruption.

Memory Usage. Memory usage for different GC targets is presented in Table 2, normalized to the vanilla ZGC with the chosen heap size as described in §5.6. Memory represents the average used memory before a GC for the last 5 stable iterations. Despite comparing memory maximums, normalization yielded similar results. Results show that overall memory usage decreases if the tested GC target is higher than the default GC CPU overhead. For instance, Hazelcast_100 has, by default, a GC CPU overhead of 21 %. Therefore, the memory used grows with 5 %, 10 %, and 15 % GC targets but is on par for a 20 % GC target. The biggest observed reduction is 96 % for Sunflow with 15 % and 20 % GC targets.

Moreover, the reduction in memory usage correlates with a higher number of minor and major collections, which simply means that GC works more to keep a tighter heap. As expected, in terms of reducing memory footprint, 20 % leads to the smallest heap size across all the benchmarks.

Execution Time. The results show that adjusting the heap size dynamically with 15 % and 10 % GC targets had a minimal negative impact on execution time, except for Xalan and Sunflow. However, Avrora, Hazelcast_400, and Fop have a reduction in execution time. For instance, Avrora showed a 3 % and 5 % improvement in execution time for 15 % and 20 % GC targets, respectively. This improvement can be attributed to the collector compacting live objects close together, improving cache locality [35], and making memory accesses easier to prefetch. However, Sunflow experienced a significant 15 % degradation in execution time. Additional profiling revealed more stalling in the instruction pipeline backend, which is often an indication of memory stalls [20]. It is possible that the 96 % memory reduction resulted in too many GC cycles, which interfered with the mutator accesses. To improve our technique in the future, we will consider the cache effects of too many GC cycles. From prior work, we also know that Sunflow is very sensitive to keeping allocation order during relocation and it is possible that this order is kept less well with so frequent GC cycles.

Energy. As per our initial hypothesis (Table 2), we expected energy changes to exhibit an opposite trend to memory. We anticipated that if a benchmark consumed more CPU during GC than the baseline, then we would see a decrease in memory usage and an increase in energy consumption. This is because CPU usage incurs higher energy costs than DRAM [18]. As expected, the 20 % GC target yields on average worse energy results compared to 10 % and 15 % GC targets. However, it is apparent that the relationship between reduced memory and increased energy is not always linear. For instance, the Graphchi benchmark with 20 % GC target has a 82 % reduction in memory usage but only 1 % increase in energy consumption. At a single-program granularity, opting for high GC targets has an increased energy cost. However, in a cloud setting, where CPU is typically highly overcommited [27] and memory is the limiting factor for consolidating virtual machines and containers, significant memory reductions lead to fewer physical nodes and ultimately lower energy consumption [2].

Latency. In our evaluation, latency results were available only for a subset of benchmarks, which we report in Table 3. Our adaptive approach has no negative impact on 99th-percentile latency and can even reduce it. For instance, in

Table 2. Execution time, memory, energy (all three normalized), the number of minor and major collections as well as GC CPU overheads in vanilla ZGC and adaptive ZGC for various benchmarks (BMs). Heap size (MB) (Z) is the minimum stall-free heap size for each benchmark. CPU Utilised shows the number of CPU cores used by the application (out of 32 cores available on SandyBridge). White cells show no statistical significance according to the methodology explained in §5.4. Different shades of red represent highlights where the adaptive approach is worse than the default. Darker red indicated the CV above 5 %. We write (Z) for vanilla ZGC and (A) for our adaptive approach.

	GC targets	*Avrora*	*Biojava*	*Graphchi*	*Hazelcast_100*	*Hazelcast_250*	*Hazelcast_400*	*Luindex*	*Spring*	*Sunflow*	*Tomact*	*Xalan*	*Fop*	*Zxing*
Memory	5	0.76	0.8	0.22	1.51	1.65	1.9	0.86	0.52	0.09	0.72	0.66	1.41	0.6
	10	0.67	0.82	0.2	1.69	1.18	1.73	0.74	0.15	0.05	0.37	0.33	0.93	0.56
	15	0.66	0.43	0.19	1.36	0.93	1.15	0.61	0.14	0.04	0.44	0.32	0.92	0.49
	20	0.65	0.18	0.18	1.03	0.82	0.9	0.54	0.13	0.04	0.48	0.31	0.86	0.43
Execution Time	5	1.01	1.01	1.04	1.01	0.98	0.96	2.05	1.05	1.17	1.0	1.09	1.04	1.02
	10	0.99	1.01	1.02	1	1	0.96	1.02	1.05	1.15	1.02	1.06	0.61	1.01
	15	0.97	1.01	1	1	0.99	0.97	1.01	1.05	1.16	1.04	1.15	0.43	1.01
	20	0.95	1.02	1	1	0.99	1.01	1.02	1.05	1.15	1.06	1.15	0.44	1.02
Energy	5	0.99	0.62	1.01	0.95	0.95	0.95	1.84	0.99	1.12	1.0	1.08	1.35	1.02
	10	0.98	0.83	1	0.95	0.99	0.96	1.12	1.09	1.14	1.02	1.08	0.74	1.02
	15	0.96	0.82	1	0.93	1.01	0.97	1.09	1.13	1.15	1.03	1.15	0.64	1.02
	20	0.93	0.8	1.01	0.95	1.02	1.01	1.15	1.19	1.14	1.05	1.17	0.66	1.03
Minor (Z)		0	142	11	1213	443	798	4296	356	38	315	146	41	9
Minor (A)	5	82	174	450	151	170	195	12412	1187	38	482	676	51	18
	10	84	175	657	136	192	239	16134	6503	1269	3783	949	50	24
	15	146	428	1649	202	437	493	18979	9730	1529	6394	1335	62	38
	20	221	3338	2790	560	782	885	25941	14239	1661	10308	1649	82	44
Major (Z)		44	20	90	108	49	57	2952	16	7	64	9	7	3
Major (A)	5	86	25	126	38	35	31	3931	36	7	67	89	23	3
	10	88	23	240	33	37	34	5987	213	90	350	117	23	7
	15	95	27	330	49	55	48	6813	375	157	576	167	26	8
	20	91	45	383	86	69	68	9804	572	202	932	215	30	11
GC CPU OH (%) (Z)		1	3	1	21	12	17	4	4	0.2	3	3	9	2
GC CPU OH (%) (A)	5	5	5	3	10	9	10	5	5	5	5	5	5	6
	10	11	11	6	11	10	12	10	10	10	11	10	11	10
	15	16	13	9	15	15	15	17	15	14	15	15	16	17
	20	21	19	11	20	21	20	22	20	18	21	20	20	18
Heap Size (MB) (Z)		1024	8192	16384	2048	4096	4096	256	4096	16384	2048	1024	256	2048
CPU Utilised/32		0.9	1.07	5.45	15	19.2	27	1.25	12.32	30.22	21.06	21	2.5	21

CPU-intensive workloads such as Tomcat and Hazelcast_400, where there is high competition for CPU resources between the collector and mutator threads, lower GC targets (i.e., 10 %) lead to using more memory and positively affect latency by allowing GC to run less frequently, thereby reducing the impact on mutator performance. While increasing *Xmx* could achieve a similar effect, our approach reduces latency while also decreasing memory usage. Because, with the fixed memory level, GC CPU overhead can vary drastically throughout execution, leading to high numbers. Our technique keeps the GC CPU overhead more stable, aiming to fluctuate around a certain GC target. It ensures that GC does not take up a lot of space, allowing mutators to deliver stable low latency without frequent drops. With a 20 % GC target and the smallest heap size, Spring showed a notable increase in latency. However, it is important to note that this benchmark has less than half of the capacity of the machine's average CPU utilization, but at times it spikes quite high, becoming CPU

Table 3. The 99th-percentile metered latency from the adaptive approach normalized to vanilla ZGC. The color coding is the same as in Table 2. H is for Hazelcast.

	Target	Tomcat	Spring	H_100	H_250	H_400
	10	0.68	1	0.4	0.91	0.69
latency	15	0.84	1.06	0.47	0.96	0.74
	20	0.96	1.18	0.73	1.14	1.16

intensive. Higher GC targets in CPU-intensive workloads can reduce latency by mitigating contention between GC and mutator threads, as explained above. However, due to the limited number of latency-oriented workloads tested and the high variance in DaCapo benchmarks, we cannot make definitive conclusions about the positive impact of our technique on latency. Nonetheless, our findings suggest that it does not have a statistically significant negative effect.

Picking the Default GC Target. Different GC targets can yield opposite trends for different optimization goals. While the highest GC target of 20 % provided the best results for memory, energy optimization requires the lowest GC target. Meanwhile, too many or too few GC cycles can harm performance. Thus, choosing the best GC target for each program may require manual selection. However, upon examining the benchmarks as a whole, we found that a 15 % GC target achieved a 51 % memory reduction, with only a 3 % execution time degradation and a 3 % increase in energy (calculated as the geometric mean across all benchmarks, following [12]). Therefore, a 15 % GC target may be a good default choice for optimizing the trade-off between memory usage, execution time, and energy consumption.

7 Related Work

Language runtimes that host managed languages—such as Java, Python, and JavaScript—maintain a garbage collected heap to manage live application objects (unreachable objects are collected by the garbage collector). Determining the heap size is challenging as it involves a tradeoff between application pause time, GC CPU, and memory utilization. Various heuristics have been proposed to achieve this goal by minimizing pause time, GC utilization, and memory usage.

7.1 Heap Size Adjustment Algorithms

A number of studies have been conducted for STW collectors, aiming to improve execution time [5], avoid paging [15] or both [37]. Brecht et al. [5] propose an adaptive technique to increase the heap size aiming to reduce execution time in the STW Boehm-Demers-Weiser GC [4]. The authors suggest increasing the heap size aggressively without collecting garbage if sufficient memory is available. Only when memory is scarce GC becomes more frequent, and the heap size stabilizes. This approach prioritizes reducing the GC target overhead to improve the application throughput.

Yang et al. [2004] introduced an analytical model to adjust the heap size in the multi-program environment. In their approach, an operating system's virtual memory manager monitors an application's memory allocation and footprint. Then, it periodically changes the heap size to closely match the real amount of memory used by the application. A model is used to minimize GC overhead by giving it enough heap size but also to minimize paging by avoiding large heaps. The model is offered for Appel and semi-space collectors [1]. Zhang et al. [39] propose a novel approach to memory management called Program-level Adaptive Memory Management (PAMM). PAMM uses the program's repetitive patterns (phases) information to manage memory adaptively. The authors believe the behavior of the phase instances is quite similar and repetitive, so they can represent the memory usage cycle in the application. PAMM monitors the program's current heap usage and the number of page faults to adjust a softbound as a GC threshold. When the threshold is reached, PAMM triggers GC to collect and free unused memory. They evaluate PAMM with three STW and generational collectors (Mark-Sweep, CopyMS, and GenCopy). PAMM relies on a specific phase detection algorithm, which may not be applicable to all types of programs.

Grzegorczyk et al. [15] propose the Isla Vista heap size adjustment strategy to avoid GC-induced paging. Their strategy is to grow the heap when more physical memory is available and shrink it by triggering GC when there is not enough physical memory. Thus, it trades more GC for less paging by communicating between OS and VM and triggering the heap size adjustment logic on relocation stalls.

Controlling the ratio of GC time to overall execution time also has been addressed within HotSpot's collectors, using `-XX:GCTimeLimit`[5]. However, this strategy may not be suitable for concurrent collectors, as they are designed to operate concurrently with the mutator and outside of the program's critical path.

In a closely aligned study, White et al. [34] propose a PID (Proportional-Integral-Derivative) controller that monitors GC overhead (the percentage of total execution time spent on GC) and adjusts the heap resize ratio to maintain a target GC overhead level set by the user. They utilize the Jikes Research Virtual Machine (RVM), the Memory Management Toolkit (MMTk) as the experimental platform, and the FastAdaptiveMarkSweep collector.

More recently, Bruno et al. [6] propose a vertical memory scalability approach to scale JVM heap sizes dynamically. To do this, the authors introduce a new parameter: `CurrentMaxMemory`. Contrary to the static memory limit defined at launch time, `CurrentMaxMemory` can be re-defined

[5]https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gc-ergonomics.html

at run-time, similar to our soft max capacity. In addition to the new dynamic limit, this work also proposed an automatic trigger to start heap compaction whenever the amount of unused memory is large. This technique allows returning memory to the Operating System as soon as possible.

7.2 Heap Size Adjustment in State-of-the-Art GC

Immix [3] is a collector suitable for high-performance computing. Immix does not require the maximum heap size to be known in advance. It continuously monitors the amount of free memory available in the heap and adjusts memory allocation accordingly. When the amount of free memory falls below a certain threshold (which may vary between implementations), Immix triggers a GC cycle to reclaim unused memory. If the free space is still insufficient after collection, Immix may allocate additional memory blocks to meet the application's memory needs. Immix also considers the rate of object allocation as a metric. If the allocation rate exceeds a certain threshold, it indicates a high memory consumption and the potential need for more memory to avoid out-of-memory errors. Immix also uses heuristics to estimate the size of the working set or the set of objects that are actively being used by the application. Since Immix is a STW collector, this dynamic heap resizing brings many disadvantages. For example, the application may experience brief pauses or slowdowns during the resizing process, which in turn makes it more difficult to reason about the memory usage and performance characteristics of an application.

Cheng et al.[8] introduce a parallel, concurrent, real-time garbage collector for multi-processors. GC work is proportional to the allocation rate, so it indirectly scales up and down with program CPU utilisation. It aims to provide bounds on pause times for GC while also scaling well across multiple processors. Using the concept of Minimum Mutator Utilisation (MMU), they capture the percentage of time in a given time window the mutators have access to the CPU. They showed that their proposed collector keeps higher MMU results compared to non-incremental GC. However, they do not assess GC CPU or utilize MMU-based actions.

Degenbaev et al. [9] propose scheduling GC during detected idle periods in the application to reduce GC latency. It uses knowledge of idle times from Chrome's scheduler to opportunistically schedule different GC tasks like minor collections and incremental marking. This allows adapting GC based on real-time application behavior and available idle cycles. While not directly adjusting heap size, scheduling GC during idle periods allows for reducing memory usage and footprint when the application becomes inactive and based on the real-time needs of the application.

The G1 [10] (Garbage First) garbage collector requires knowledge of the maximum memory needed for an application in advance. If it is not explicitly provided, it uses a default value. G1 uses a dynamic heap size adjustment strategy to adjust the memory usage during runtime based on the current usage pattern of the application [28]. G1 divides the heap into regions of equal size and groups them into two generations: young and old. When the young generation fills up, G1 performs a young collection, during which live objects are copied to a new region while unused regions are reclaimed. G1 also performs periodic concurrent marking of live objects in the old generation. When the old generation fills up, G1 performs a mixed collection, which collects both young and old regions that have been marked as garbage. During a mixed collection, G1 dynamically sizes the heap by using the occupancy of the old generation as a target and adjusts the heap size to meet that target.

Heap size adjustment in .NET is difficult because of the prevalence of object pinning which can make it impossible to uncommit memory. .NET offers a ConserveMemory interface to the garbage collector that allows "conserving memory at the expense of more frequent garbage collections and possibly longer pause times" [32]. This setting works by controlling the fragmentation tolerance in old generations, before triggering a full, compacting GC cycle.

7.3 Discussion

Previous works, adjust the amount of heap size by estimating the amount of memory that is necessary to keep the application running without incurring high latency and CPU overheads. Instead of estimating the amount of memory needed by the application, we adjust the heap size to meet a specific GC target. Our CPU-driven heap size adjustment is particularly important for concurrent collectors like ZGC, which compete with the mutator for CPU resources to collect memory, unlike the STW collectors used in prior studies. Dissimilar to fixed-size heap headroom used in STW collectors, a concurrent GC requires variable headroom depending on the available CPU for collection. If the mutator consumes most of the CPU, a large headroom is necessary for a concurrent collector, while a small headroom suffices for collection when the mutator has minimal CPU usage. In sum, rather than directly controlling the heap headroom as in previous works for STW collectors, we specify the desired GC target and adjust the heap headroom accordingly.

8 Conclusion

This paper explores an adaptive approach for automatically adjusting heap size based on CPU overhead for GC work as a tuning knob. Our evaluation demonstrates that this technique does not negatively impact latency, which is the main goal of fully concurrent collectors. In addition, we offer insights into optimizing energy and performance by tuning GC targets. Our ongoing work focuses on seamlessly integrating and refining this approach within the ZGC framework to unlock its full potential in real-world applications.

References

[1] A. W. Appel. 1989. Simple Generational Garbage Collection and Fast Allocation. *Softw. Pract. Exper.* 19, 2 (feb 1989), 171–183.

[2] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take It to the Limit: Peak Prediction-Driven Resource Overcommitment in Datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 556–573. https://doi.org/10.1145/3447786.3456259

[3] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *ACM SIGPLAN Notices* 43, 6 (2008), 22–32.

[4] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[5] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2001. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Sigplan Notices* 36, 11 (2001), 353–366.

[6] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic vertical memory scalability for OpenJDK cloud applications. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 59–70.

[7] Zixian Cai, Stephen M Blackburn, Michael D Bond, and Martin Maas. 2022. Distilling the real cost of production garbage collectors. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 46–57.

[8] Perry Cheng and Guy E Blelloch. 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 125–136.

[9] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. 2016. Idle time garbage collection scheduling. *ACM SIGPLAN Notices* 51, 6 (2016), 570–583.

[10] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[11] Ben Evans. 2020. *What Tens of Millions of VMs Reveal about the State of Java*. Retrieved Feb 28, 2023 from https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/

[12] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29 (1986), 218–221.

[13] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, et al. 2021. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. 14, 12 (2021), 3110–3121. https://doi.org/10.14778/3476311.3476387

[14] Frank E Grubbs. 1969. Procedures for detecting outlying observations in samples. *Technometrics* 11, 1 (1969), 1–21.

[15] Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. 2007. Isla vista heap sizing: Using feedback to avoid paging. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 325–340.

[16] Jun Han and Claudio Moraga. 1995. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *From Natural to Artificial Neural Computation*, José Mira and Francisco Sandoval (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 195–201.

[17] Matthew Hertz and Emery D. Berger. 2005. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/1094811.1094836

[18] Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[19] Intel. 2009. *Intel Architecture Software Developer's Manual*. Vol. Volume 3: System Programming Guide.

[20] Intel. 2020. *Intel VTune Profiler Performance Analysis Cookbook: Top-down Microarchitecture Analysis Method*. Retrieved April 25, 2023 from https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html

[21] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall.

[22] Stefan Karlsson. 2022. *Z Garbage Collector*. Retrieved Feb 28, 2023 from https://wiki.openjdk.org/display/zgc/Main

[23] Stefan Karlsson. 2023. *JEP draft: Generational ZGC*. Retrieved Feb 28, 2023 from https://openjdk.org/jeps/8272979

[24] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. 2022. Optimal Heap Limits for Reducing Browser Memory Use. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 160 (oct 2022), 21 pages. https://doi.org/10.1145/3563323

[25] Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (1983), 419–429.

[26] Jonas Norlinder, Erik Österlund, and Tobias Wrigstad. 2022. Compressed Forwarding Tables Reconsidered. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 45–63. https://doi.org/10.1145/3546918.3546928

[27] OpenStack. 2022. *Overcommitting CPU and RAM*. Retrieved April 28, 2023 from https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html

[28] Oracle. 2021. *Garbage-First Garbage Collector Tuning*. Retrieved April 23, 2023 from https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-garbage-collector-tuning.html#GUID-3D3E4662-1E89-42EE-96FA-836C0E7C97AA

[29] Semih Sahin, Wenqi Cao, Qi Zhang, and Ling Liu. 2016. Jvm configuration management and its performance impact for big data applications. In *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 410–417.

[30] Marina Shimchenko, Mihail Popov, and Tobias Wrigstad. 2022. Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/3546918.3546925

[31] Marko Topolnik. 2020. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*. Retrieved May 10, 2022 from https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch

[32] Genevieve Warren, Maoni Stephens, Sébastien Ros, GitHubPang, Andrew Au, and Peter Sollich. 2023. *Runtime configuration options for garbage collection*. Retrieved April 28, 2023 from https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector#conserve-memory

[33] Bernard L Welch. 1938. The significance of the difference between two means when the population variances are unequal. *Biometrika* 29, 3/4 (1938), 350–362.

[34] David R White, Jeremy Singer, Jonathan M Aitken, and Richard E Jones. 2013. Control theory for principled heap sizing. *ACM SIGPLAN Notices* 48, 11 (2013), 27–38.

[35] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving Program Locality in the GC Using Hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/3385412.3385977

[36] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (sep 2022), 34 pages. https://doi.org/10.1145/3538532

[37] Ting Yang, Matthew Hertz, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. 2004. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th international symposium on Memory management*. 61–72.

[38] Karen K Yuen. 1974. The two-sample trimmed t for unequal population variances. *Biometrika* 61, 1 (1974), 165–170.

[39] Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. 2006. Program-level adaptive memory management. In *Proceedings of the 5th international symposium on Memory management*. 174–183.