Improving relocation performance in ZGC by identifying the size of small objects

JINYU YU

Master's Programme, Embedded Systems, 120 credits
Date: May 23, 2022

Supervisors: Tobias Wrigstad, Per Lidén, Erik Österlund, Thilanka Thilakasiri
Examiner: Matthias Becker
School of Electrical Engineering and Computer Science
Host company: Oracle
Swedish title: Förbättrad omplaceringsprestanda i ZGC genom att identifiera storleken på små objekt

Abstract

Modern Garbage Collectors provide performance improvements by increasing program locality to utilize the faster CPU cache. A common approach is to move objects together according to the mutators' access order, which brings more relocations during GC. In most cases, more relocations would not impact performance when using concurrent Garbage Collectors such as ZGC. However, in constrained environments with fewer CPU cores or less memory, bad relocation performance will cause overall performance degradation. In this thesis, we investigated why larger objects do not benefit from better program locality, then proposed a new design to reduce the number of relocations by efficiently identifying and ignoring larger objects. As a result, the relocation performance can be improved. In constrained environments, this can lead to an increase in overall throughput.

In the new design, we introduce an extra page type, the tiny page. If an object is considerably small that it could benefit from relocation, it will be placed on the tiny page when allocating. As a result, we could replace the time-consuming size check of objects with a faster page type check. Memory fragmentation also can be reduced by this design.

To evaluate this design, we add the size identification procedure into a locality improvement implementation named HCSGC. The results of benchmarks show a slight improvement in constrained environments. In the JGraphT benchmark, we see a 3-5% speedup in different configurations with memory limitations. In the SPECjbb2015 benchmark, we see a ~1% increase in performance on average, but with overlapping confidence intervals. In the DaCapo benchmark suite, we see a 1% improvement in the sunflow benchmark with CPU constraint. For other benchmarks in DaCapo, no significant difference is discovered. The results suggest that the proposed new design is a feasible way of filtering out larger objects, and doing so can further improve the relocation and overall performance.

Keywords

Garbage Collector, Java, Data locality

Sammanfattning

Modern Garbage Collector ger prestandaförbättringar genom att öka programplatsen för att använda den snabbare CPU-cachen. En vanlig metod är att flytta fler objekt baserat på mutators åtkomstorder. I de flesta fall skulle fler omplaceringar inte påverka prestanda vid användning av samtidiga Garbage Collector som ZGC. Men i begränsade miljöer med färre CPU-kärnor eller mindre minne kommer dålig flyttningsprestanda att leda till övergripande prestandaförsämring. I denna avhandling undersökte vi varför större objekt inte gynnas av bättre programplats, och föreslog sedan en ny design för att minska antalet flyttningar genom att effektivt identifiera och ignorera större objekt. Som ett resultat kan flyttningsprestandan förbättras. I begränsade miljöer kan detta leda till en ökning av den totala genomströmningen.

I den nya designen introducerade vi en extra sidtyp, den lilla sidan. Om ett objekt är avsevärt litet som kan ha nytta av omplacering, kommer det att placeras på den lilla sidan vid allokeringen. Som ett resultat kan vi ersätta den tidskrävande storlekskontrollen av objekt med en snabbare sidtypskontroll. Minnesfragmentering kan också reduceras med denna design.

För att utvärdera denna design lägger vi till storleksidentifieringsproceduren i en implementering av lokaliseringsförbättring som heter HCSGC. Resultaten av riktmärken visar en liten förbättring i begränsade miljöer. I JGraphT-riktmärket ser vi en hastighet på 3-5% i olika konfigurationer med minnesbegränsningar. I riktmärket SPECjbb2015 ser vi i genomsnitt en ~1% prestationsökning, men med överlappande konfidensintervall. I DaCapo -riktmärket ser vi en förbättring på 1% i solflödesriktmärket med CPU-begränsning. För andra riktmärken i DaCapo upptäcks ingen signifikant skillnad. Resultaten tyder på att den föreslagna nya designen är ett genomförbart sätt att filtrera bort större objekt, och det kan ytterligare förbättra flytten och den övergripande prestandan.

Nyckelord

Garbage Collector, Java, Datalokalitet

Acknowledgments

I would dedicate my thesis to everyone who has helped or supported me during the writing.

First of all, I would like to thank Oracle and Uppsala University for providing me a great opportunity to involving in the OpenJDK project. A special thanks to Oracle for maintaining such a comprehensive open-source project that benefits billions of people. It is an honor for me to contribute to the brilliant project.

I would like to express my gratitude to Prof. Tobias Wrigstad for all his help. Without his valuable guidance, there is no way to start working on such a complicated project. I could not thank him enough for all the thoughts and suggestions he provided.

I would like to thank my company supervisors Erik Österlund and Per Lidén for their willingness to help and professional instructions during the meetings.

I would like to thank Albert Yang for his great work on HCSGC, which makes it possible for me to implement my optimization and write this thesis.

I would like to thank my examiner Prof. Matthias Becker and KTH supervisor Thilanka Thilakasiri, for all the feedback they provided. Without their feedback, the outcome of this thesis would not have been at this level.

Finally, I am thankful to my family and friends for their constant love and encouragement.

Stockholm, May 2022
Jinyu Yu

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

CMS Concurrent Mark Sweep

EC evacuation candidate

GC Garbage Collector

HBIR High Bound Injection Rate

HCSGC Hot-Cold Objects Segregation GC

JDK Java Development Kit

JIT Just-In-Time

JRE Java Runtime Environment

JVM Java Virtual Machine

OOP object-oriented programming

RT Response-Throughput

STW Stop The World

TLAB Thread Local Allocation Buffer

ZGC The Z Garbage Collector

Chapter 1

Introduction

The Garbage Collector (GC) plays a critical role in managed programming languages such as Java. Memory is considered garbage if the heap-allocated records are not reachable by any chain of pointers from program variables [1]. Instead of release the unused memory manually, the runtime of managed languages provides GC to automate this process. By using GCs, the developer does not need to take care of object lifetimes or match allocations with deallocations. This will completely eliminate some errors related to memory management, including the double-free problem and most types of memory leaking [2].

The ease of garbage collection comes with the cost of some additional runtime overhead, which impacts the program's overall performance. Two key goals are set for GCs to minimize the impact, throughput and latency. Throughput is a measurement of workload within a specific unit of time, higher throughput results in higher efficiency of the program. Latency is the responsiveness of the application. The GC thread needs to pause all mutator threads, or so-called "Stop The World (STW)", to reclaim unused memory. It will make the application unresponsive for some time, therefore latency is usually measured by the maximum pause time.

Modern GC attempts to improve the overall performance of runtime virtual machines using several techniques. First, to improve the latency, the parallel collection is implemented by deploying multiple collector threads, including the ParallelGC [3] and the Concurrent Mark Sweep (CMS) GC [4]. However, this is useless for applications with very large heap since the pause time can be several

seconds or more. Then, concurrent collectors, such as The Z Garbage Collector (ZGC) [5] or Shenandoah GC [6], are developed to deal with the large heaps. They can drastically reduce the STW time by making GC threads running alongside mutator threads. Also, to improve the throughput, other approaches are proposed to increase the program locality by moving objects around[7, 8, 9]. A better program locality can bring a higher cache hit rate, the throughput will increase as memory access time reduces.

The object reordering process increases the relocation count in each GC cycle and sometimes even increases the count of total GC cycles. In most cases, more relocation will not impact the overall performance for concurrent GCs, as the relocation runs on a different thread from the mutator thread and not likely to block the program in over-provisioned environments. However, this could be different in constrained environments. Poor relocation performance may affect the overall performance in two ways.

- It may aggravate the memory fragmentation and increase latency. When the allocation rate is higher than what GC can keep up with, or more specifically, if the heap is exhausted before the GC finishes, the mutator thread will enter *Allocation Stall* phase and will be blocked until the end of the current GC cycle. More relocations will increase GC time and lead to a longer *Allocation Stall*. Also, longer GC time means a lower GC capability, which brings more *Allocation Stall*s. We could trade throughput for latency by assigning more CPU-time to GC with the `-XX:ConcGCThreads` parameter [2], but this may not be an option in constrained environments.

- Poor relocation performance will also lower the overall throughput. In constrained environments, GC may not have idle cores to run on, and it will have contention with mutator threads for CPU time. As a result, any GC performance regression will cause an overall throughput regression.

A. M. Yang proposed Hot-Cold Objects Segregation GC (HCSGC) [8], a design to classify objects to *hot* and *cold* objects and reorder the objects based on the mutators' access order. The design will increase the program locality significantly but introduces more relocations. In HCSGC, the relocation amount is reduced by filtering out *cold* objects

Page Size Class	Page Size	Object Size
Tiny	2 MB	[0, 256] B
Small	2 MB	(256B, 256kB]
Medium	32 MB	(256 KB, 4 MB]
Large	$N \times 2(> 4)$MB	> 4 MB

Table 1.1: Tiny Page Size Class

and only relocate *hot* objects. It also ignores the Medium pages and Large pages and only improves locality for objects in small pages. However, even in small pages, the object size could be large and no locality benefits to be reaped from the better locality. Thus, filtering out and moving only the objects which are sufficiently small *(tiny objects)* should lead to an increased return on investment.

In this thesis, we proposed a new design that efficiently identifies the size of small objects. We also extended HCSGC to relocate the tiny objects only to further increase the relocation performance without affecting the overall performance.

1.1 Problem and Purpose

Modern GC tries to improve the overall performance of runtime virtual machines by moving objects around. However, the object reordering process may take a long time and lead to performance regression in constrained environments. We propose that ignoring the larger objects and moving only the objects which are sufficiently small (tiny objects) could increase the relocation performance as well as the overall performance.

We present several ways of optimizing the relocation phase. First, we propose the tiny page method, as shown in Table 1.1. The three page classes in ZGC are further extended to four page classes. With the newly introduced tiny page class, we can decide whether relocate or not in the allocation phase. This design also helps reducing memory fragmentation in HCSGC. Second, a naive size check method is also implemented. This method could work with the tiny pages to reduce the memory fragmentation, or without the tiny pages, to reduce overhead in allocation.

Research Question 1:
Do bigger objects gain benefits from better locality? What is the threshold of object size that start to gain benefits?
Research Question 2:
Which one has better performance of checking size of objects, the tiny pages or the naive size check?

1.2 Goals

The goal of the degree project is to propose a new design that efficiently identifies the size of small objects for ZGC. The new design would not bring regressions in over-provisioned environments, and is supposed to have an overall performance improvement in constrained environments.

- Subgoal 1:

  Develop the proposed methods of identifying the size of tiny objects. The introduced code should pass sanity check (i.e. not crashing the Java Virtual Machine (JVM) and produce right results).

- Subgoal 2:

  Tune the threshold of *tiny* objects. Find the size that start to gain benefits from locality.

- Subgoal 3:

  Benchmark the methods with different algorithms. Find if any of them cause regression in over-provisioned environments.

- Subgoal 4:

  Benchmark in constrained environments and find improvements in overall performance.

1.3 Scope

A viable and efficient size check for a specific platform and CPU architecture should work for other platforms and CPU architectures to some extent. Although Java is developed for platform independence,

port the design to all platforms is not in the scope of this thesis. All implementations and performance tests are based on the Linux system and x86_64 CPU architecture.

1.4 Ethics and Sustainability

The project has no ethical risks. All data we collected is non-sensitive, and no personal information is used. We use only benchmark software to produce data, which means we have no physical or social risks during data collection, and all data is reproducible. The OpenJDK and the underlying ZGC are open-source under the GPLv2 license[10]. It means we can freely modify and redistribute the code and the software as long as we disclose the modification under the same license. Thus, the project is facing no legal issues.

The project aims to increase the relocation performance by reducing the number of objects to be relocated. It results in a lower CPU and memory consumption and possibly better performance in low-power devices. As a result, the project can help reduce power consumption and be eco-friendly.

1.5 Structure of the thesis

Chapter 2 presents relevant background information about OpenJDK and GC, as well as the HCSGC on which the thesis is based. Chapter 3 first argues that large objects do not benefit from a better locality, then presents a high-level description of different ways to identifying object size, followed by the detail of actual implementations. Chapter 4 covers the evaluation methodology, and chapter 5 shows the evaluation results with discussions. Finally, chapter 6 concludes and presents potential future work.

Chapter 2

Background

2.1 OpenJDK

OpenJDK is an open-source implementation of Java. It is used daily by millions of people and thousands of businesses, such as the leading smartphone operating system Android. As is used by mission-critical industrial applications, the OpenJDK project only accepts the highest-quality contributions. Also, the project contains tens of millions of lines of code, and some parts of the code even came from more than 20 years ago, making it a delicate process to contribute to OpenJDK [11].

OpenJDK provides Java Development Kit (JDK) for compiling and debugging Java files, and Java Runtime Environment (JRE) for running the compiled Java programs. JVM, the main component in JRE, provides the bytecode interpreter and Just-In-Time (JIT) compiler to run the program, as well as several different GCs for a better tuning strategy.

2.1.1 The Interpreter and JIT Compiler in OpenJDK

One of Java's great strengths is that you can write your code once and then it runs on every system that supports the JVM. Instead of being compiled into a specific binary for a specific CPU, the Java code is compiled into an idealized assembly language, Java bytecode. Then the bytecode is run by the JVM. This gives Java the platform independence of an interpreted language [12].

But the interpreted code will almost always be measurably slower

than compiled code. To resolve this, JVM is able to compile the bytecode into the platform binary as the code executes. The compilation occurs as the program is executed, so the compiler is called Just-In-Time (JIT) compiler [13]. The JVM provides two kinds of JIT compilers, the `C1` compiler and the `C2` compiler. The `C1` compiler, which is also called the `Client` compiler, does less optimization to the code and compiles much faster than the other one. The name `Client` also imply that the compiler is used in the relatively slower "client" hardware, although it's not really true today, 20 years after the term was utilized. The `C2` compiler also called the `Server` compiler or `Opto` compiler, does more optimizations and much slower to compile.

The primary difference between the two compilers is their aggressiveness in compiling code. The `C1` compiler begins compiling sooner than `C2`, so it will be faster at the beginning of code execution, while the other compiler gains while it waits. *Tiered Compilation*, which is enabled by default in OpenJDK, will make the code first compile by the `C1` compiler. When a code section is executed frequently, it is recompiled by the `C2` compiler. The `C2` compiler uses the `C1` compiler and the interpreter to generate compiled versions of methods that collect profiling information about themselves. The compiled code is substantially faster than the interpreter, and the program executes with greater performance during the profiling phase. *Tiered Compilation* can also achieve better peak performance than a regular `C2` compiler since it has a longer period of profiling, which can yield better optimization [14].

2.1.2 Different Garbage Collectors in OpenJDK

Garbage Collectors make assumptions about the way applications use objects, therefore the choice of GCs can affect performance to a great extent. OpenJDK provides several different GCs and also deprecates or removes some of the old GCs in newer versions. Current available GCs are listed below [2].

- Serial Garbage Collector

  The serial collector uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads. It's best-suited to single processor machines or applications with

small data sets. It can be explicitly enabled with the option `-XX:+UseSerialGC`.

- Parallel Garbage Collector

  The parallel collector is also known as *throughput collector*. It's a generational collector similar to the serial collector. The primary difference is that the parallel collector has multiple threads that are used to speed up garbage collection. It can be enabled with the option `-XX:+UseParallelGC`.

- The G1GC - Garbage first garbage collector

  G1 is a mostly concurrent collector. It perform some expensive work concurrently to the application. This collector is designed to scale from small machines to large multiprocessor machines with a large amount of memory. It provides the capability to meet a pause-time goal with high probability, while achieving high throughput. G1 is introduced in JDK7 and used to replace the deprecated CMS GC. G1 is selected by default on most hardware and operating system configurations, or can be explicitly enabled using `-XX:+UseG1GC` .

- The Z Garbage Collector

  ZGC is a non-generational, mostly concurrent, parallel, mark-compact, region-based scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads. ZGC provides max pause times of a few milliseconds, but at the cost of some throughput. It is intended for applications which require low latency. Pause times are independent of heap size that is being used. ZGC supports heap sizes from 8MB to 16TB [5].

  ZGC is included in OpenJDK releases since JDK11. In JDK 11, it can be enabled by the special flags `-XX:+UnlockExperimentalVMOptions` and `-XX:+UseZGC`. In later versions, the experimental flag can be omitted.

- Shenandoah Garbage Collector

  Shenandoah is a low pause time GC that reduces GC pause times by performing more garbage collection work concurrently with the running Java program. Shenandoah does the bulk of GC

work concurrently, including the concurrent compaction, which means its pause times are no longer directly proportional to the size of the heap [6].

Shenandoah availability differs by vendor and JDK release. OpenJDK 12+ builds normally include Shenandoah by default. OpenJDK 11 requires the opt-in during build time. It can be enabled by JVM parameter `-XX:+UseShenandoahGC`.

- Epsilon: A No-Op Garbage Collector

  Epsilon handles memory allocation but does not implement any actual memory reclamation mechanism. It intends to provide a completely passive GC implementation with a bounded allocation limit and the lowest latency overhead possible, at the expense of memory footprint and memory throughput. As a no-op GC, the Epsilon GC is mainly used in extremely short lived jobs, or for test purpose such as performance testing, memory pressure testing and VM interface testing [15].

  The Epsilon GC can be enabled by `-XX:+UseEpsilonGC`.

2.2 Garbage Collection and Object Allocation

An object is considered garbage and its memory can be reused by the VM when it can no longer be reached from any reference of any other live object in the running program. From the view of GC, a program consists of two kinds of threads, one or more *mutator* threads and one or more *collector* threads. The *mutator* threads contain several GC roots, which are origin points to reach the rest of the objects in the heap. If there is a path to an object from the root, the object is a *live* object. Otherwise, the object is unreachable and considered garbage.

2.2.1 Generational Heap Space

For most of the generational GCs, the heap space is divided into young and old generations. When the young generation fills up, a *minor collection* will be invoked and only do garbage collection within the young generation. Eventually, the old generation fills up and must be collected, resulting in a *major collection*, in which the entire

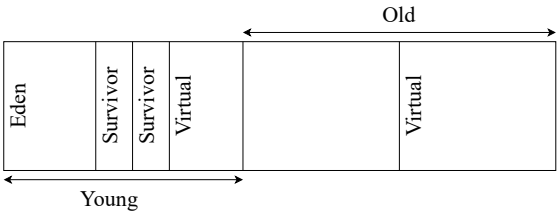


Figure 2.1: The heap space of a generational GC

heap is collected. Major collections usually last much longer than minor collections because a significantly larger number of objects are involved. The young generation consists of Eden and two Survivor spaces. Most objects are initially allocated in Eden. When running garbage collection, objects in Eden and Survivor space will move to the other Survivor space. When copied a certain number of times or there is no space left, the objects will be moved to the old generation. The heap space and different regions are shown in Figure 2.1.

In JVM, the Garbage Collector will also handle the allocation of objects. For example, although the no-op GC Epsilon does not implement any actual "garbage collection" functions, it handles memory allocation. It is easier to get control over the object lifecycle if the GC handles both allocation and deallocation.

2.2.2 Thread Local Allocation Buffer

In Java, most new objects are allocated in Eden, which is a space shared between threads. If multiple mutator threads try to allocate new objects at the same time, synchronization will take place for these threads, which will definitely slow down the program. To deal with the contention between threads, Thread Local Allocation Buffer (TLAB) is introduced. TLAB is a small region in Eden that is exclusively assigned to a thread. Since each thread can only write to its own TLAB, there is no need for synchronization. TLAB is enabled by default, and can be disabled by explicitly using the `-XX:-UseTLAB` parameter[16].

If TLAB is enabled, when a thread try to allocate an object, it will instead allocate a much larger space (around 1% of Eden space) as TLAB, and then allocate the object inside the TLAB. Each TLAB contains three pointers, `start`, `top` and `end`. The `start` and `end` pointer identifies the position of TLAB in Eden, which prevents other

thread using the space. The `top` pointer, which equals to `start` when TLAB is created, identifies the position for the next allocation. The allocations in TLAB are done by bumping pointers, or called "Bump pointer allocation". More specifically, when a thread try to allocate an object, the pointer to the new object will be `top`, and the `top` pointer itself will add by the size of the new object, so the memory space between the new `top` pointer and old `top` pointer is reserved to the new object. Part of the TLAB allocation related code [17, threadLocalAllocBuffer.inline.hpp:35-55] is shown in Listing 2.1.

```
inline HeapWord* ThreadLocalAllocBuffer::allocate(size_t size
    ) {
  HeapWord* obj = top();
  if (pointer_delta(end(), obj) >= size) {
    // successful thread-local allocation
    // This addition is safe because we know that top is
    // at least size below end, so the add can't wrap.
    set_top(obj + size);

    return obj;
  }
  // no enough space inside current TLAB, fallback to slower
   allocation
  return NULL;
}
```

Listing 2.1: TLAB Allocation Code

Allocations inside TLAB is actually located on the Eden, so the TLAB can be directly discarded when it is full, then objects will be left on the Eden space. However, it could be a hard thing to tell whether a TLAB is full. When the object that is going to be allocated is bigger than the available space in TLAB, we can either discard the TLAB and allocate the object in a new TLAB, or allocate the object in Eden and wait for a smaller object. Too many discards will cause memory fragmented and increase the memory foorprint; too many Eden allocations will reduce the effect of TLAB and increase contention. In OpenJDK, the choice is done by using a waste limit. If the object cannot be placed in the TLAB and the available space is smaller than the waste limit, the TLAB will be discarded and the object will be put into a new TLAB. If the available space is larger than the waste limit, the object will do its allocation in the Eden space.

A more detailed decription of TLAB allocation flow is shown Fig-

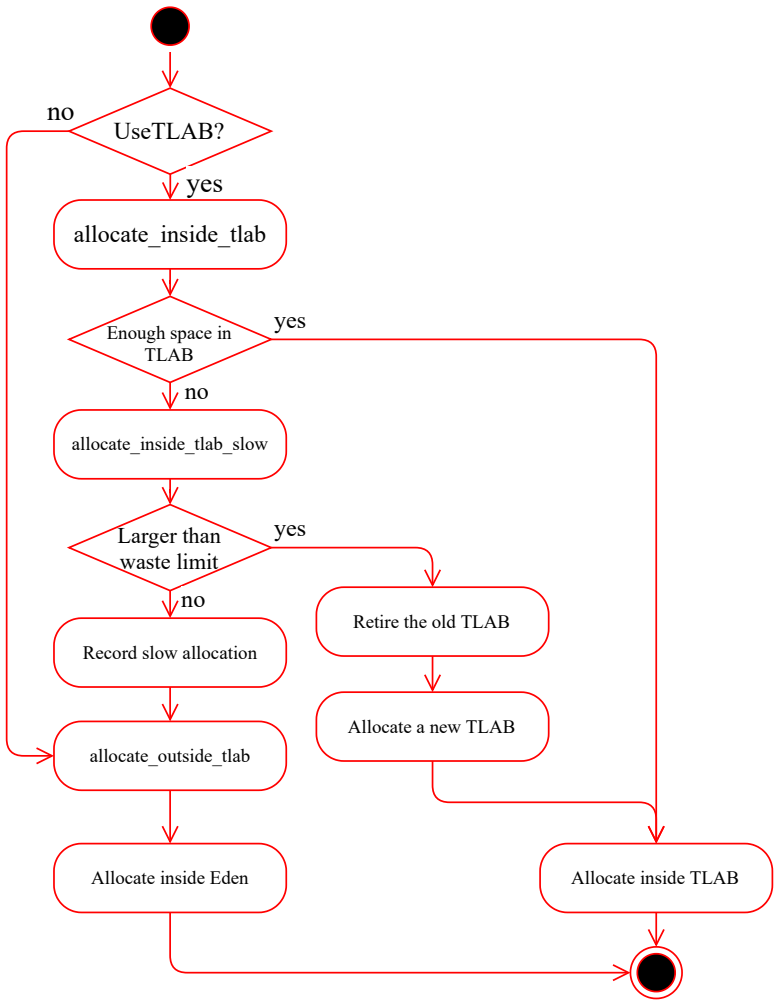


Figure 2.2: TLAB Allocation Flowchart

ure 2.2. When allocation happens, the fast path, `allocate_inside_tlab` method in `MemAllocator` will be invoked, and code shown in Figure 2.1 will try to allocate the object inside TLAB. If the TLAB cannot hold the new object, a `NULL` pointer will be returned, and the `allocate_inside_tlab_slow` will run instead. The slow path will first check if the amount free in the TLAB is too large to discard. If the free space in TLAB is larger than the waste limit, the TLAB will be retained, and a `NULL` pointer will be returned, then the `allocate_outside_tlab` will be invoked to allocate the big object in the Eden space directly. If the free space in TLAB is smaller than the

Page Size Class	Page Size	Object Size
Small	2 MB	[0, 256] KB
Medium	32 MB	(256 KB, 4 MB]
Large	$N \times 2 (> 4)$ MB	> 4 MB

Table 2.1: ZGC Page Size Classes

waste limit, the TLAB will be retired and a new TLAB will be created.

The waste limit is set by `-XX:TLABRefillWasteFraction`, the fraction between the waste and total size of the TLAB. By default, this value is 64, which means the TLAB will be discarded if the available space is less than $1/64$ of the TLAB size.

2.3 The Z Garbage Collector

ZGC is a non-generational, region-based, mostly concurrent, parallel, mark-compact garbage collector. It is included in OpenJDK releases since JDK11 and is available for all platforms since JDK14 [5]. ZGC uses multiple-memory mappings, colored pointers, and load barriers to enable concurrent garbage collection work.

ZGC is a single generation GC. Unlike G1GC, ZGC has no identification of age of objects, so in every GC cycle, ZGC marks all live objects. Instead, ZGC puts all objects into different page regions, also called ZPage, which can be dynamically allocated and sized into one of the three sizes shown in Table 2.1. A bump-pointer allocation scheme as shown in Figure 2.1 is used for small and medium pages. When the current page can not satisfy the requested size, a new page is allocated. An object larger than 4 megabytes will always have its own page.

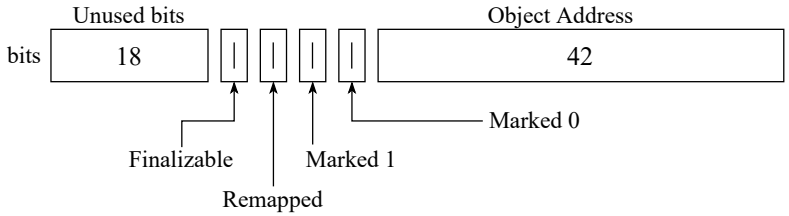


Figure 2.3: Colored pointer in ZGC

The “Colored Pointers” is one of the core concepts of the ZGC

implementation. Every pointer in ZGC is colored with different higher-order bits, which contains metadata for the pointer. Four bits are taken into action currently, *Finalizable*, *Remapped*, *Marked0*, and *Marked1*. The positions of these four bits are shown in Figure 2.3. One problem that arises with colored pointer is that some pointer masking mechanism should be implemented to mask out the four-color bits. In some architecture such as ARM, the pointer masking is hardware supported, after setting the mask, operations will automatically ignore the specified bits. However, the x86 architecture does not support this mechanism in hardware. Instead, multi-mapping is proposed. Multi-mapping is a concept that maps multiple ranges of virtual memory to the same physical memory. In ZGC, only one of *Remapped*, *Marked0*, and *Marked1* bits could be 1 at any given time, therefore three mappings should solve the pointer mask issue.

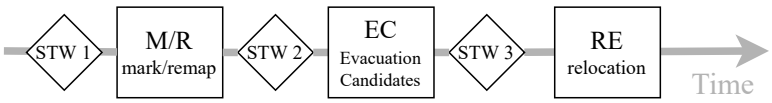


Figure 2.4: ZGC cycle with three STW pauses

ZGC is designed as a low latency GC, which achieves STW time independent of the heap size and never exceeds 10 milliseconds. As shown in Figure 2.4, one ZGC cycle has three STW pauses and three concurrent phases.

A ZGC cycle starts with a short STW pause **STW1** that decides which color is the good color by alternating two colors. If *Marked0* is selected as good color, then *Marked1* will be good color at the next ZGC cycle. After the good color is decided, all GC roots will be marked to good and push to mark stacks. Followed by STW1 is the classical mark/remap phase **M/R**, which runs concurrently. If the GC finds any pointer with a bad color, the pointer will be updated to the current address and marked to good color. Also, the size of all living objects on a page is recorded to decide if a page is sparsely populated and needs evacuation. **STW2** is merely a synchronization point that ensures all marking is completed. Then the **EC** phase will begin and select all pages that do not have enough living objects using the previously collected liveness information. The selected page is referred to as the

relocation set. In **STW3**, the good color changes to *Remapped*, and all roots pointing into EC are relocated and have good color. In **RE**, all live objects in EC will be relocated.

2.4 The HCSGC

A. M. Yang proposed HCSGC [8] based on ZGC. The new design dynamically reorganizes objects and attempts to place objects in the way they are brought into the cache when they are about to be used by the mutator. If an object is brought into cache due to access by a particular mutator thread, the subsequent accesses by the same thread usually hit the adjacent object. Many changes have been implemented to achieve the reorganize, including revised EC selection, deferring relocation phase, speculative hot-cold segregation, and operating only on small pages.

Revised EC selection. The original criteria to select EC in ZGC is the live ratio of the page, but more objects should be selected in order to get a better locality. HCSGC proposed *hot* and *cold* objects as well as *weighted live bytes* to perform a intelligent EC selection. The *hot* and *cold* classification avoid extra relocation for objects that have no gain in locality. HCSGC introduced a new command-line parameter `-XX:+UsePartialEvacuation` to enable this.

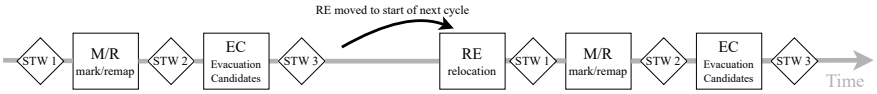


Figure 2.5: Deferring relocation phase

Deferring Relocation phase. Mutators and GC threads are competing to relocate objects on EC pages. The original ZGC cycle shown in Figure 2.4 is aimed to perform most of the relocation on GC threads. As a result, the mutator threads can read the updated address in the forwarding table, thus, lowering the loads on mutator threads. However, with the hotness information, the object layout will be rearranged, which could bring more relocation overhead. Therefore, HCSGC defer the relocation phase as shown in Figure 2.5. As a result, objects will be relocated according to the access pattern between two

GC cycles, which increases the locality. This could be enabled by the parameter `-XX:+UseLazyRelocate`.

Speculative hot-cold segregation. This design segregate hot and cold objects in memory, to further increase the chances of cache-friendly placement. By using the hotness information of each object, during the relocation phase the hot and cold objects will be moved to different destinations in memory.

Operating Only on Small Pages. Large objects could be easily over-approximated as hot objects according to the scheme of HCSGC, which might cause the reduction of cache effectiveness. Therefore, HCSGC only deals with small pages and leaves medium and large pages intact. However, the size for small pages is still quite large for the scheme, and a new smaller page size class is required to maximize the benefits of HCSGC. We aim to solve this in our proposal.

2.5 Related works

People tried to utilize the cache by focusing on the cache locality from many years ago. Robert C. presented an adaptive memory management algorithm to increase the locality [18]. He proposed that dividing the address space into regions characterized by generation, volatility, and activity can improve the locality of objects. This algorithm introduced the *train-space*, besides the *from-space*, *scavenge-space*, and *new-space* in usual copying garbage collectors. The *train-space* is very similar to the *from-space* and is used to identify the activity of objects.

Lam et al. showed an approach to improve the locality by considering the type and format of objects and then grouping co-active objects together with effective heuristics [19]. Object type directed function grouping and data structure grouping are proposed in the thesis. Authors assume that the object's type indicates the access pattern, so the data tag information of the object is used as the key to group related objects. They also proposed that the most commonly used data type in Lisp, the *cons* cell, could be used by garbage collectors to recognize trees or association lists, then the data structure can be guessed and grouped.

Huang et al. presented the online object reordering that detects the program traversal patterns to improve spatial locality [7]. The

proposal utilized the copying phase of a generational collector to put hot objects before cold ones. Then topological locality could be improved by putting hot fields together with their parent.

Chen et al. proposed an online profile-guided proactive approach to improve the locality [9]. The work collects object access information by a low-overhead mechanism and improves both cache locality and page locality. This method could run independent of GC cycles and is more flexible.

Unfortunately, most of the proposals tried to improve locality on a generational garbage collector. At the time of writing the thesis, the generational heap is not implemented in ZGC yet.

Chapter 3

Design and Implementation of Object Size Identification

In object-oriented programming (OOP) languages such as Java, programs' data consist of objects, and the contents of fields in an object are references. One drawback of this is that OOP tends to have a lot of pointer chasing, which brings more memory access if these references are distributed over a large memory range, causing degradation in overall throughput. So locality is important for any memory-bounded computation. In languages like C/C++, a better locality can be achieved by carefully placing the objects. But in managed languages, the memory layout is changed frequently in the process of garbage collection. An optimal object layout designed by programmers could easily be destroyed. As a result, the locality problem can only be solved in the GC.

The technique in HCSGC will improve locality for small objects, also filtering out the cold objects to avoid relocate too many objects to affect the performance. The design can be further improved by only selecting tiny objects to relocate. In HCSGC, all hot objects on small pages will be relocated, even if the object is so large that could not benefit from a better locality. We propose that ignoring these large objects will improve the relocation performance but not impact the speedup from the locality.

In this chapter, we will first discuss why the performance of OOP languages rely heavily on locality, followed by why bigger objects are not affected by locality. Then, we propose two types of object size identification method, including a naive method which checks the

object size in the relocation phase, and a “tiny page” method. The “tiny page” method will mark the size class of an object by putting smaller objects to tiny pages when it is allocated. After that, implementations will be described, including the details in the interpreter and two JIT compilers.

3.1 Object Size and Locality

The locality of objects could be classified as the locality inside one object and the locality between several objects. The object itself is stored sequentially, so the first type of locality is relatively easier to achieve. There is no point in increasing the inner locality of small objects, as they can fit in the cache line as a whole. For large objects, inner locality can be improved by merely hardware optimizations without insights on the actual running code. This is done by partitioning data into blocks that can fit into a single cache line, then using some simple prefetching method such as next-line prefetching [20]. On the contrary, the locality between objects is hard to improve considering the impossibility of predicting accessing patterns.

3.1.1 Locality inside one object

Large objects benefit from a better locality inside themselves. Consider two loops that iterating a large array, as shown in Listing 3.1. The first loop multiplies every element with 3, while the second loop only changes every 16th element. Although the second loop only does 1/16 of the calculations, it takes roughly the same amount of time to run [21].

```
for (int i = 0, n = array.length; i < n; i++) {
    array[i] *= 3;
}

for (int i = 0, n = array.length; i < n; i+=16) {
    array[i] *= 3;
}
```

Listing 3.1: Two simple loops

The reason behind this is the difference in the locality. A typical integer is 4 bytes, and the minimum amount that transfers between memory and cache, also called cache-line size, is 64 bytes. Thus, a cache-line can store 16 consecutive integers in an array. In the first loop, the CPU fetches 16 integers into the cache and does the calculations with all 16 numbers from the cache. In the second loop, the CPU also needs to fetch the 16 integers into the cache, but only calculate the first number from the cache. Memory access time is much slower than calculations in modern CPUs, and both loops fetch and stored the same amount of data from the memory, as a result, they take a similar amount of time.

3.1.2 Locality between objects

Objects also benefit from a better locality between them. The contents of fields in objects are accessed with a pointer-chasing style and might be fragmented in memory. Therefore, the locality between objects plays a critical role in improving the performance of accessing fields.

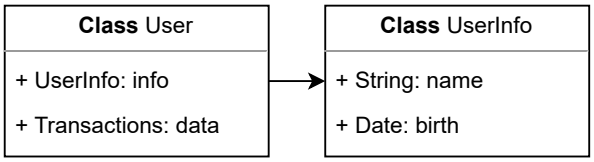


Figure 3.1: Simple pointer-based objects

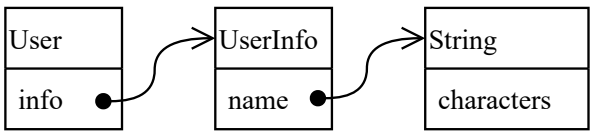


Figure 3.2: Access pattern of the demo object

Consider a simple user management system that stores user information in the form of Figure 3.1. When trying to get the `name` field from a `User` object, we need to go through the path `user.info.name`, which contains two pointer operations as shown in Figure 3.2. If the three objects are located in different places in memory, the CPU needs to access the memory three times to get the

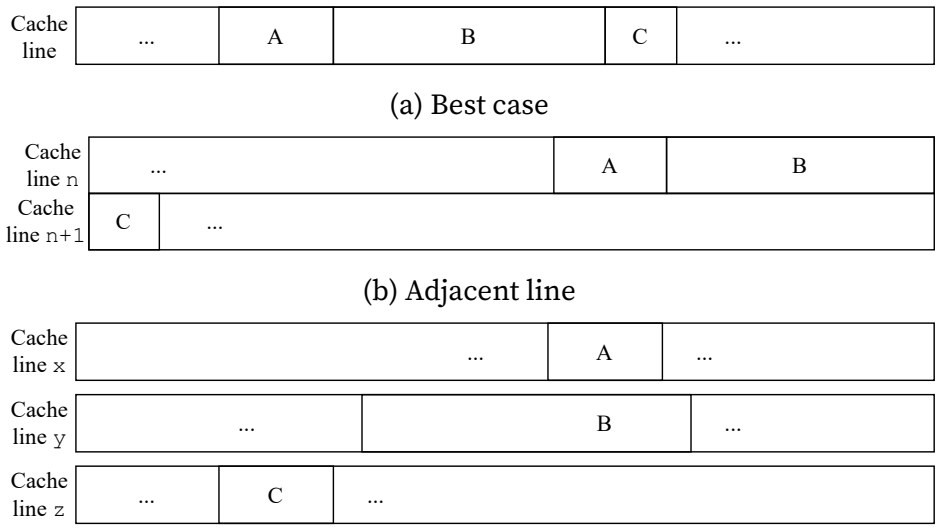


Figure 3.3: Locality for very small objects

required data. With a better locality, these pointers may be stored sequentially and fall in a single cache line, which means the data can be fetched by a single access.

3.1.3 Moving objects to improve locality

Moving objects together could improve the locality and sometimes leads to better overall performance. Small objects that could be loaded into the cache as a whole are likely to benefit from this. If we do not consider runtime optimization such as software cache prefetching, only objects that can fall in the same cache line can gain from the locality. Any access to an object will make the CPU fetch the whole line of objects into the cache, which acts as a hardware prefetching.

Considering three small objects A, B, C, that will be accessed in order. The worst locality we can get is to place them randomly, as shown in Figure 3.3c. Three accesses of memory are needed to get all three objects. To improve the locality, these objects can be rearranged and put together. In the best case, they can fall in the same cache line (as shown in Figure 3.3a). Accessing A will bring all three objects to the same cache line, speed up the following accesses. However, these objects could fall in adjacent lines (Figure 3.3b), which might bring another memory access. In modern processors, various hardware

prefetching mechanisms are implemented. The next-line prefetcher, a basic prefetch mechanism that will fetch the next cache line(s) after a cache demand, has been implemented in most processors such as AMD's [22] and Intel's [23]. With the prefetcher, access of object A will bring the first line to cache, and in the meanwhile, emit a prefetch for the second line. The following accesses will have a speed up similar to the previous case. Therefore, moving small objects that could fall in the same cache line can improve the locality and the performance.

However, improving the locality by moving objects does not always result in better performance, especially for large objects that could not fit in a single cache line. The reasons are as follows.

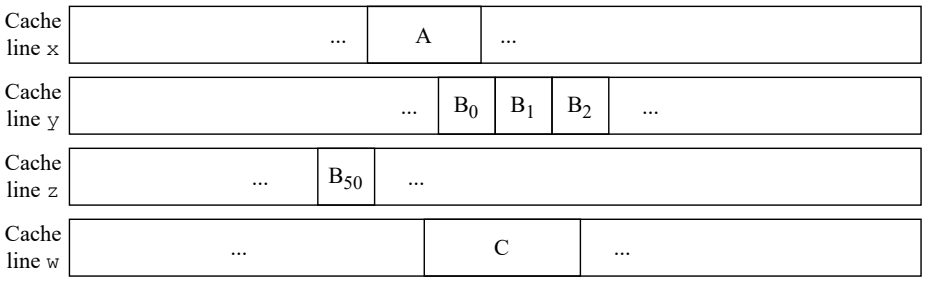
- Moving objects will not change the locality inside themselves. For larger objects like arrays, most of the accesses are iterating over the object. Thus, the locality and performance are mainly determined by the iteration patterns and inner layout. The location of the object will not affect the iteration performance, and the layout is static and could not be changed by the runtime. As a result, moving such an object will not improve the overall performance.

- If the large objects are accessed in a pointer-chasing style, locality might be improved by moving them closer, but almost no performance benefits can yield from such a moving.

  Consider three objects A, B, C. A and C are small objects, while B is an array with 128 Integers that will take more than 500 Bytes of memory. The three objects will be accessed in the order "A-B[50]-C". The best locality we can get is shown in Figure 3.4a, and the worst locality is shown in Figure 3.4b. Since object B is larger than the cache line size, it will not load into the cache as a whole. Instead, only the cache line that contains B[50] will be loaded. It is clear from the figures that the memory address we required is far from both boundaries of object B. Thus no matter how close these objects are, they cannot fall in the same cache line and load at once. For both cases, three accesses to memory are required. Therefore, we cannot get performance improvement by moving them around.

  Moving large objects might gain a small benefit according to boundary effects. If the accessed position is among the leading

(a) Best case



(b) Worst case

Figure 3.4: Locality for larger objects

or trailing ones in the array, it has a possibility of falling in the same cache line with other objects. A better locality of such objects could improve the performance. Consider we are accessing B[0] instead of B[50] in the previous example. B[0] could fall in the same cache line with A if objects A and B are relocated and placed together, then the memory demand could decrease by one. However, this only applies to objects that are slightly larger than the cache line size whose boundary could take a significant proportion to the whole object. For objects much larger than the cache line, no noticeable improvement could yield from such few boundary positions in them.

- Furthermore, the cost of relocating large objects is high. Moving such objects will consume extra system bandwidth, introduce more contention to the memory, and decrease the overall performance. If the relocation is processed on the mutator thread, it will further slow down the overall performance by blocking the thread for a longer time.

Objects slightly larger than cache line size could get benefits from the locality in some cases. Besides the boundary effects, the hardware prefetcher could also help improve the performance of these objects.

The hardware prefetcher will load several more cache lines when there is a cache demand, making it possible for the whole object to be loaded into the cache even if the object is slightly larger than the cache line. AMD's CPU will load a maximum of 13 more cache lines [22], while Intel's will load up to 20 within the same 4KBytes page [23]. However, there are many other restrictions for the prefetcher, including the cache availability and access patterns, which makes it usually fetch much fewer lines. We still need to care for a longer time to move larger objects around either. Therefore, we must keep a balance between a better locality and a better relocation performance for these slightly larger objects.

3.1.4 Relocation in ZGC and HCSGC

In ZGC, the relocation phase executes concurrently, and most of the relocations are processed in GC threads. Relocations will not impact the overall performance in over-provisioned environments.

Things could be different in HCSGC. Several mechanics that aim to increase locality were introduced, including Enlarging EC and Lazy Relocate. They increase the relocation amount and expose mutators to more relocation. As a side-effect, they bring more workloads on the mutator and make the memory more fragmented, and in some cases, lower the overall performance.

- Lazy Relocate is used to relocate objects according to mutators' access pattern. This mechanic will defer the relocation phase of each GC (which is the last phase in original ZGC) to the start of the next GC cycle and makes it possible for mutators to relocate objects between two GC cycles. As a result, some relocations that were originally processed in GC thread will be moved to the mutator thread. These relocations will compete with other workloads and can lower the performance if the relocation set is large.

- Enlarging EC means to flag more pages as evacuation candidates to improve more objects' locality. However, it will increase the relocation amount in both mutator threads and GC threads. For the mutator part, it will further compete with the workers and bring degradation to the performance. Relocations in GC threads will not affect the performance in most cases, except

in CPU-constrained environments, where all threads themselves are competing for the limiting resource.

- Lazy Relocate will also retain the garbages longer, as the relocation phase is moved to the next GC cycle. This will cause a more fragmented memory and increase memory usage. In memory-constrained environments, this will drastically increase the number of GC cycles and result in an overall slowdown.

3.2 Relocation Performance Improvement

As described in section 3.1.4, two key implementations in HCSGC, Enlarging EC and Lazy Relocate, will lower the relocation performance. The problem can be solved from two aspects, lower the relocation amount and reducing the memory fragmentation.

According to our proposal in section 3.1.3, only objects smaller than the cache line size (64 Bytes) or slightly larger than that could benefit from the locality. We name these objects as tiny objects. The larger objects on small pages that could not benefit from the locality are named as small objects. Our designs focused on how to identify and filter out the small objects and select the tiny ones.

3.2.1 Reduce the memory fragmentation

As the small objects will be ignored, we no longer need to lazy relocate them. They can be fully relocated at the end of each GC cycles like the original ZGC does. By doing so, the garbage will not be retained to next GC anymore, thus reduce the fragmentation.

However, the full relocation is on a page basis. We cannot fully relocate some objects but lazy relocate other objects on the same page. In HCSGC, Medium and Large pages will be fully relocated, and Small pages (object size smaller than 256KBytes) will be lazy relocated.

Our threshold for tiny objects is apparently lower than the small pages limit, so we further divided the small page type to tiny pages and small pages, as shown in Table 1.1. The tiny pages will store the tiny objects, while the small page will store the small objects that could not benefit from relocation (larger than tiny objects but smaller than 256KBytes). As a result, we can safely fully relocate all small pages as

they will never be selected to lazy relocate, and memory fragmentation could be reduced.

The memory fragmentation could be further reduced by implementing a separate TLAB for tiny pages and small pages. By only accepting the tiny objects, the tiny TLAB could be filled granularly, then the waste of TLAB could be reduced. In most Java applications, tiny objects take the major proportion to the total memory, so any improvements in the utilization of tiny TLABs could result in better memory usage and less memory fragmentation.

3.2.2 Reduce the relocation amount

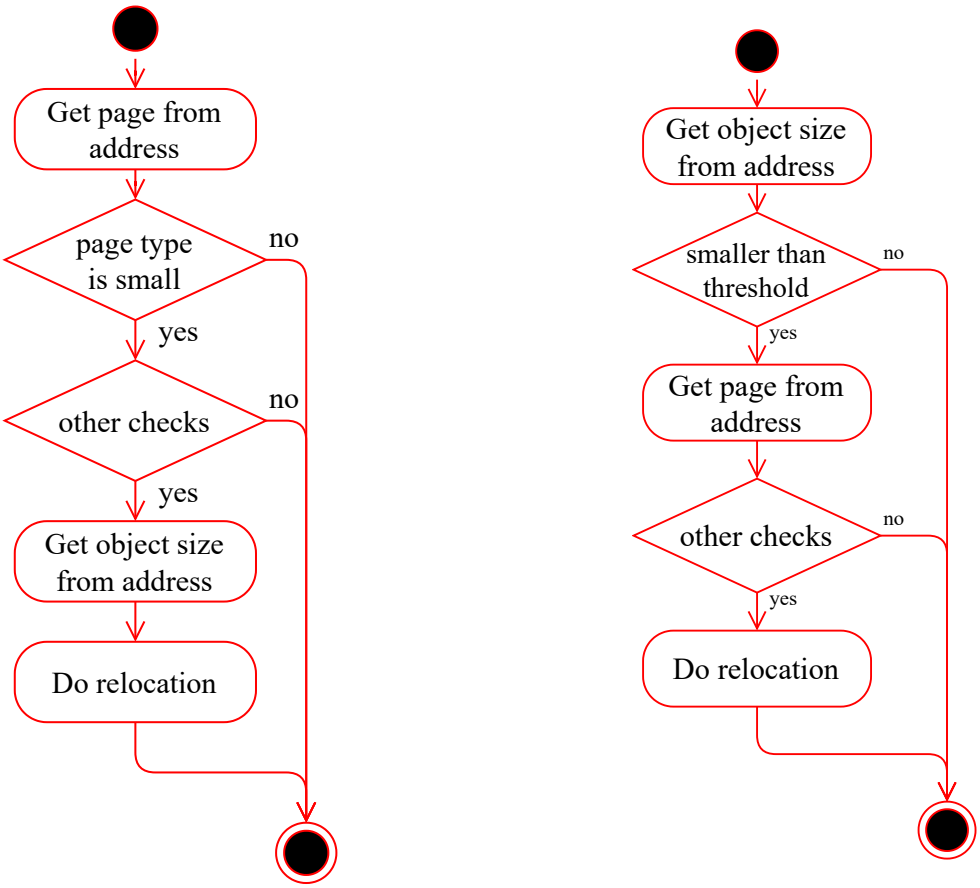
Reducing the relocation amount without harming the locality is the key to bring improvements to the overall performance. All objects on small pages (smaller than 256KBytes) will be selected for relocation in HCSGC. We proposed to select the tiny objects only but ignore the small objects, then the question comes to how to efficiently identify the tiny objects and ignore the small objects during relocation.

The selection method in HCSGC is shown in Figure 3.5a, which is mainly done by a page type check. If the object to be relocated is not on a small page, it will be ignored from the relocation. Then other checks will be handled, including whether the page is relocatable and whether the object is already relocated. If all checks passed, the size of the object will be acquired, and then relocation will be handled.

We came up with two ideas for identifying tiny objects. One is the naive method, which will check the size of each object first, ignoring the small objects and relocate the tiny ones, as shown in Figure 3.5b. The other is the tiny page method, which will keep the original selection in HCSGC, but replace the check for small pages with a check for tiny pages. The tiny page method will ignore all other pages and only select tiny pages during relocation.

Both of the proposed methods need to get the page and object size from the address. Therefore, the performances of the success path of them are the same. However, if the object is not a tiny object, the performance differs. The object size is the only information need to be acquired in the naive method, while in the tiny page method, it's the page type. Any difference between obtain the object size and page type will reveal in the relocation performance.

The code to get page type from object address is shown in Listing



(a) HCSGC

(b) Naive

Figure 3.5: Different size checks

3.2. First, the page on which the object is located will be obtained from the page table. This step contains several bitwise operations to get the page index, then dereference the page from the page table. Second, the page type is read from the corresponding field in the page object.

```
offset = address & ZAddressOffsetMask;
index = offset >> ZGranuleSizeShift;
page = page_table[index];
return page->_type;
```

Listing 3.2: Get page type from object address

The code to get object size from the address is shown in Listing 3.3. First, the address will be casted and dereferenced to a Java class object. Second, the size information of the object will be acquired. Last, the size is calculated and returned. Although it seems as simple as the previous code to get the page type, this one could use a long time to run when getting the size information.

For most of the Java classes, their size information will be computed at class initialization, so the `size()` function is merely a field access which is considerably fast. However, this does not apply to all classes. For arrays, their size could not be obtained when initialize. Instead, the element size will be computed and stored. When using the `size()` function on an array object, extra operations need to be handled, including getting the length of the array and multiplying the length with the element size.

Making things worse, we need to filter out the small objects, which are mostly arrays. Therefore, it is slower to get the size information of these objects. In the tiny page method, this step will be skipped since the small objects are not located in tiny pages; but in the naive method, their size is checked first. This will introduce a significant degradation in relocation performance.

```
oop = cast_to_oop(address);
size_in_words = oop->size();
bytes = size_in_words << LogBytesPerWord;
return bytes;
```

Listing 3.3: Get object size from address

As a result, the tiny page method which checks for page type during relocation is selected. We will discuss about the actual performance difference in Section 5.1.

3.2.3 The Tiny pages

Adding the tiny pages could reduce the fragmentation introduced by HCSGC, as well as lowering the relocation amount. The tiny pages require a separate TLAB - the tiny TLAB. As stated in section 3.2.1, the new TLAB could also reduce the memory fragmentation. Although adding a new TLAB may increase the memory footprint, the effect is minor, as it will occupy at most 2MBytes of memory per CPU core.

The default object size limit for the tiny page is set to 256 bytes. As is discussed before, this value should make a balance between locality and relocation performance. Also, this value should be larger than the cache line size (64 bytes) and smaller than the small object size limit (256 Kbytes). The best value for a tiny object size limit may vary between different programs, so we make a command-line option to set the size limit. For most of the benchmarks we tested, 256 bytes can be the best choice, so it is set as the default size limit.

The page size of the tiny page is set to 2 Mbytes, which is the same as small pages. The smallest page that ZGC can handle on an x86_64 architecture is 2 Mbytes, so we could not make a smaller page. On the other hand, a larger page will increase the memory footprint. As a result, 2 Mbytes is selected as the tiny page size. Having the same page size for the small and tiny pages might introduce some conflicts. We will further discuss this in Section 3.3.

Furthermore, the tiny page will introduce an inevitable extra check when allocating the object, which may cause performance degradation to object allocation. We will discuss the reasons and optimizations of this overhead for each implementation in Section 3.4.

An alternative way is to discard the optimization of memory fragmentation and use size check for relocation. By doing this, the relocation amount could still be reduced and the tiny pages could be exempted so no overhead for object allocations. However, this might not be an good option, since the memory fragmentation plays a dominant role in relocation performance. We will discuss about the actual performance difference in Section 5.1.

3.3 Adding a Page Size Class

The tiny page size is set by code shown in Listing 3.4, which is located in the `zGlobals.hpp` file [17]. The `ZGranuleSizeShift` is

provided by ZGC, and identifies the smallest page size it can handle. This value varies on different architectures. The granule size shift equals to `21` on x86_64, means a 2 Mbytes minimum page size.

```
const size_t ZPageSizeTinyShift = ZGranuleSizeShift;
const size_t ZPageSizeTiny = (size_t)1 << ZPageSizeTinyShift;
```

Listing 3.4: Set tiny page size

The command-line parameters are defined in the `gc_globals.hpp` file [17]. As shown in Listing 3.5, the option can be added by using the `product` macro. The first field indicates the accepted data type, the second field for option name, third for default value, and the last one for description.

```
product(size_t, ZObjectSizeLimitTiny, 256, "Objects smaller
    than this will be considered as a tiny object")
```

Listing 3.5: Add tiny object limit parameter

The `zPage.inline.hpp` file provides some utilities to work with different pages, including the `type_from_size` function, which causes a conflict between small pages and tiny pages. The function tries to return the page type based on the page size. Since small pages and tiny pages have the same size, it cannot distinguish between them only by the page size. The solution would be one of the following.

- The function is used to transform a physical memory block into the corresponding page when combining small pages to a large page or split a large page into small ones. We can record the page type when the page is stored in the memory, then, when loading the memory, read the record to decide which type of page it is.

  This is a relatively unfeasible way. The function is working with the physical memory, and we cannot add marks on physical memory. Although we could use the colored pointer to store this information, it is not practical. There are four page types currently, so two bits are needed to store the page type. This function is rarely used, and occupying several bits in the limited pointer address space seems profitless. Using a fixed bit to represent page type also results in non-extendable page types.

  Otherwise, a table that records all memory addresses and their corresponding page type is needed. However, this method will add extra overhead to page load and increase memory footprint.

- An alternate way is to regard all pages with a 2MBytes size as a tiny page. This will treat small pages as tiny pages when using this function. One typical behavior is that when we split a large page containing small and tiny objects, this function will put both of them into tiny pages.

  However, doing so will not affect the performance because it is acceptable to place small objects inside tiny pages. The tiny page and the small page are identical in most characteristics, except the lazy relocation behavior. Regarding the small pages as tiny pages will introduce an extra lazy relocation for that particular page (which is normally happening in HCSGC as it will lazy relocate all small pages). After that, these small objects will be relocated to small pages and no longer needs lazy relocation. Moreover, this function is rarely used, so this degradation that only happens once does not have a visible impact on the overall relocation performance.

Each TLAB belongs to one page class. So a TLAB for tiny objects should also be implemented. There are two ways to fulfill this:

- Track two independent TLAB in each Thread. The two TLABs have their own lifecycles, which do not affect each other.
- Extend the TLAB class to hold two sets of `top` and `end` pointers so the lifecycle can stay unchanged.

Although the second design is easier to implement, we chose the first design because it is compatible with other GCs. The new TLAB will only be enabled when using ZGC, therefore it keeps the original behaviors when using other GCs.

Since the allocations differ in interpreter and JIT compiler, they will be described in separate parts.

3.4 The Implementation of Allocation

The major part of adding a new page class is implementing the corresponding object allocation code for it. We have implemented the allocation for both inside/outside the TLAB in all three Java runtime tiers, the bytecode interpreter, the `C1` Compiler, and the `C2` Compiler.

3.4.1 Java Interpreter

The modification in the bytecode interpreter is quite straightforward. We need to check the size of the ongoing allocation and do the allocation in corresponding places. When it is inside TLAB, the allocation will be done by the tiny or small TLAB of the current thread. When outside TLAB, the request is sent to the eden heap, then allocated inside the corresponding page.

Eden Allocation

The Eden allocation is the most fundamental one in Java. As shown in Figure 2.2, if allocation in TLAB failed, it will fallback to Eden allocation. The allocation of TLABs is also handled by Eden allocation.

The Eden allocation is controlled by the `ZObjectAllocator` class. When an object is going to be allocated, its size will be passed to the `alloc_object` function, and then the place will be reserved in corresponding pages.

The selection of pages is determined by a simple if...elseif...else conditional statement. The tiny object could be checked by adding a tiny check before the original small object check, as shown in Listing 3.6. This extra check will not bring visible performance degradation to allocation. The vast majority of objects are tiny objects, which always succeeds in the first condition check. Comparing the original allocator, larger objects need one more check, but they will not impact the performance notably due to their rareness.

```
if (size <= ZObjectSizeLimitTiny) {
    return alloc_tiny_object(size, flags);
} else if (size <= ZObjectSizeLimitSmall) {
    return alloc_small_object(size, flags);
} else if (size <= ZObjectSizeLimitMedium) {
    return alloc_medium_object(size, flags);
} else {
    return alloc_large_object(size, flags);
}
```

Listing 3.6: Eden allocation

A special case is the allocation of the tiny TLABs. The allocation of TLABs is controlled by the Eden allocator, and the tiny TLABs needs to be placed on tiny pages since they contain tiny objects. However, the tiny TLAB itself is larger than the threshold of tiny objects. Therefore,

the allocation of TLABs must be handled separately. For tiny TLABs, the underneath `alloc_tiny_object` will be invoked directly for allocation.

Inside-TLAB Allocation

Most of the allocations in Java are handled by TLABs. We add one field `_use_tiny_tlab` in `MemAllocator`, as shown in Listing 3.8. The field is initiated when constructing the memory allocator. When allocating, the word size of the object will be passed to the allocator. From that, we can decide whether the object is a tiny object. The code shown in Listing 3.7 can choose the right TLAB for allocation and let the TLAB do the allocation. The TLAB allocation is unmodified as shown in Listing 2.1.

```
ThreadLocalAllocBuffer& tlab = _use_tiny_tlab ? _thread->
    tlab_tiny() : _thread->tlab();
HeapWord* mem = tlab.allocate(_word_size);
```

Listing 3.7: TLAB Allocation in Interpreter

```
const bool _use_tiny_tlab;

MemAllocator(Klass* klass, size_t word_size, Thread* thread)
  : _thread(thread),
  _klass(klass),
  _word_size(word_size),
  _use_tiny_tlab(UseZGC && (word_size <= ZObjectSizeLimitTiny
    / HeapWordSize))
{ }
```

Listing 3.8: Use Tiny TLAB Cache in MemAllocator

Considering the performance, our proposal will introduce an extra size check when allocating objects inside TLABs, or more specifically, when constructing the `MemAllocator`. This check brings regression but is necessary for placing objects to the right TLAB. Furthermore, a new `MemAllocator` is needed for each allocation inside TLABs, so all regressions in the constructor will reflect in object allocations.

The TLAB allocation handled by the Java interpreter is slow and impossible to optimize, so we implement the new TLAB allocation in JIT compilers. When JIT is enabled, most of the allocations will be handled by the native code generated by JIT (the rest will fall back to

the Eden allocator), which is considerably faster. Therefore, TLAB allocations handled by the interpreter could be greatly reduced to avoid performance degradation.

3.4.2 C1 Compiler

In the JIT Compilers, the program will be compiled into native code to improve its performance. The `C1` Compiler is easy to implement as it only generates the assembly but does not optimize it. The `C1` Compiler has two stages when doing the allocation. The `try_allocate` function in the frontend will check the parameters and then send the allocation task to corresponding backends. Then, backends, which contains `tlab_allocate` and `eden_allocate`, do the allocations.

The frontend of allocation

The original allocation code in `C1` Compiler is shown in Listing 3.9.

```
void C1_MacroAssembler::try_allocate(Register obj, Register
    var_size_in_bytes, int con_size_in_bytes, Register t1,
    Register t2, Label& slow_case) {
  if (UseTLAB) {
     tlab_allocate(noreg, obj, var_size_in_bytes,
    con_size_in_bytes, t1, t2, slow_case);
  } else {
     eden_allocate(noreg, obj, var_size_in_bytes,
    con_size_in_bytes, t1, slow_case);
  }
}
```

Listing 3.9: Original Frontend C1 Allocation

The size of object is passed by two parameters: `Register var_size_in_bytes` and `int con_size_in_bytes`. When allocating an array, the array length is unknown in the compilation time. Therefore the object size could only be passed by a register that stores the actual size of the array. In this case, the `con_size_in_bytes` will equals to `0`. When allocating an object, the actual size is fixed. The size is passed by the integer, and the register `var_size_in_bytes` will equal to `noreg`, which means a null register. The flowchart of the modified frontend check is shown in Figure 3.6.

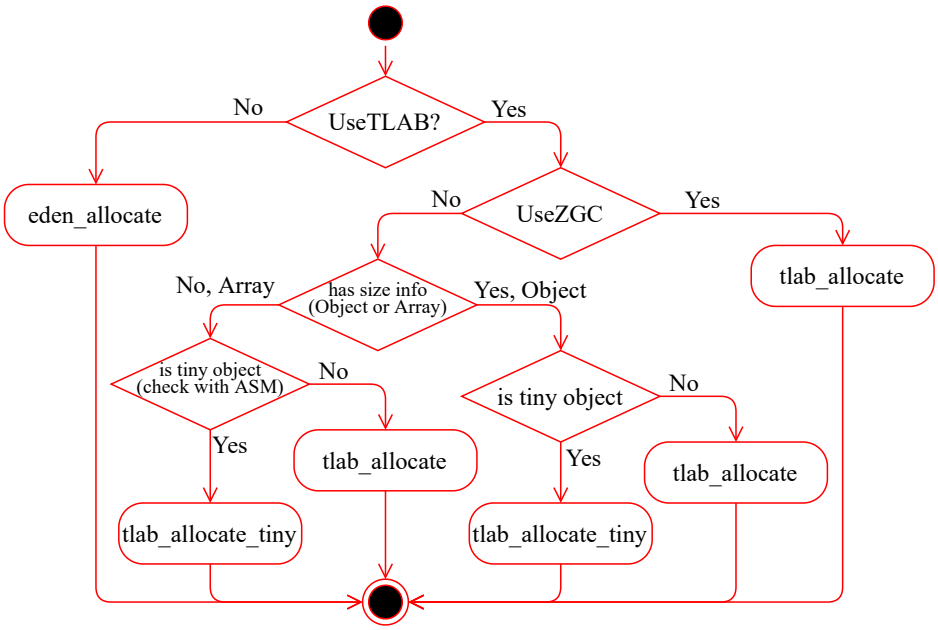


Figure 3.6: Modified Frontend C1 Allocation

```
if (UseTLAB) {
  if (UseZGC && (var_size_in_bytes == noreg) && ((size_t)
   con_size_in_bytes <= ZObjectSizeLimitTiny)) {
    tlab_allocate_tiny(...);
  } else if (UseZGC && (con_size_in_bytes == 0)) {
    Label normal;
    Label end;
    cmpl(var_size_in_bytes, ZObjectSizeLimitTiny);
    jcc(Assembler::greater, normal);
    tlab_allocate_tiny(...);
    jmp(end);
    bind(normal);
    tlab_allocate(...);
    bind(end);
  } else {
    tlab_allocate(...);
  }
} else {
  eden_allocate(...);
}
```

Listing 3.10: Modified Frontend C1 Allocation

Note: All allocators has the same parameters:
`(noreg, obj, var_size_in_bytes, con_size_in_bytes, t1, t2, slow_case)`

In the modified code, we use the parameters to decide whether allocate the object inside the TLAB of small pages or tiny pages. If the allocation is for an object with a known size, it can be decided by a normal check on `con_size_in_bytes`. Otherwise, an extra branch is needed. The `C1` Compiler provided several macros to generate the corresponding assembly, including the `cmpl` to compare two numbers, the `jcc` to jump based on conditions, and `bind` to create tags. By using these macros as shown in Listing 3.10, we can easily create a conditional branch that handles the allocation in different TLABs.

The backend of allocation

```
verify_tlab();

movptr(obj, Address(thread, tlab_tiny_top_offset()));
if (var_size_in_bytes == noreg) {
  lea(end, Address(obj, con_size_in_bytes));
} else {
  lea(end, Address(obj, var_size_in_bytes, Address::times_1))
    ;
}
cmpptr(end, Address(thread, tlab_tiny_end_offset()));
jcc(Assembler::above, slow_case);

movptr(Address(thread, tlab_tiny_top_offset()), end);
// recover var_size_in_bytes if necessary
if (var_size_in_bytes == end) {
  subptr(var_size_in_bytes, obj);
}
verify_tlab();
```

Listing 3.11: Tiny TLAB Allocation in C1 (pseudo code)

The backend of allocation in `C1` Compiler contains `tlab_allocate` and `eden_allocate` that do the allocation in TLAB and Eden respectively. The `tlab_allocate` is not changed, and it still does the allocations in TLAB of small pages. The `eden_allocate` will invoke the underlying allocations in `zCollectedHeap`, which is already adopted to check the size in the previous part. The only modification is the newly introduced `tlab_allocate_tiny` that allocate inside the tiny TLAB as shown in Listing 3.11. The code will directly move the object to the `top` pointer of the TLAB and increase the `top` pointer.

The tiny object allocation implementation in the C1 Compiler has a low but inevitable impact on the performance. In the original code, the size check is handled implicitly by putting all objects into the TLAB. Medium and Large objects cannot fit in the TLAB, so only small objects can be successfully allocated by the fast path and enter the TLAB. However, after adding the new page type and TLAB, both tiny and small objects can fit in each other's TLABs, so we must add an extra check to make sure they go to the right one.

Nevertheless, the extra size check only has a small impact because it only applied for arrays, which take a small part in objects. Performance of allocating other objects remains the same since the same amount of instructions are generated. The modified allocation backend also has the same routine as the original one. As a result, the performance degradation may be invisible.

3.4.3 C2 Compiler

The major work of the C2 Compiler is similar to C1, which is getting the top and the end pointer of the TLAB and moving the object. The difference is C2 will have optimizations, so not assembly, but grammar trees are constructed. To achieve this, different Nodes are created and add to the grammar tree. After that, the tree will be optimized by pruning and combining branches, also make predictions to different branches to further improve the efficiency.

```
void PhaseMacroExpand::set_eden_pointers(Node* &eden_top_adr, Node* &
    eden_end_adr) {
  if (UseTLAB) {
    Node* thread = transform_later(new ThreadLocalNode());
    int tlab_top_offset = in_bytes(JavaThread::tlab_top_offset());
    int tlab_end_offset = in_bytes(JavaThread::tlab_end_offset());
    eden_top_adr = basic_plus_adr(top()/*not oop*/, thread, tlab_top_offset);
    eden_end_adr = basic_plus_adr(top()/*not oop*/, thread, tlab_end_offset);
  } else {
    CollectedHeap* ch = Universe::heap();
    address top_adr = (address)ch->top_addr();
    address end_adr = (address)ch->end_addr();
    eden_top_adr = makecon(TypeRawPtr::make(top_adr));
    eden_end_adr = basic_plus_adr(eden_top_adr, end_adr - top_adr);
  }
}
```

Listing 3.12: Get Allocation Pointers in C2

The original code that gets the pointer of the TLAB is shown in Listing 3.12. Same to C1, the slow path allocation in Eden is handled

by the underlying `zCollectedHeap`, so we only care about the TLAB part. Several helper functions are used in the original code:

- `transform_later`:
  Since the `C2` Compiler is a process of building grammar trees, leaf nodes should be recorded and add to the tree. The `transform_later` function is used to record the node provided in the argument.
- `in_bytes`:
  A simple cast to mitigate different integer lengths in different platforms.
- `basic_plus_adr`:
  Will create an `AddPNode`, a node used to add a pointer with an integer.
- `MakeConX`:
  A macro that create an const long integer node `ConLNode`.

We add an extra parameter to the function `Node* size_in_bytes` to convey the size of the object. Also, since the tiny TLAB is only available in ZGC, we introduce an extra check. The overall structure of the modified code is shown in Listing 3.13. Notice this check will not introduce overhead to the program since the compiler code will only be executed once to generate the assembly.

```
void PhaseMacroExpand::set_eden_pointers(Node* &eden_top_adr,
    Node* &eden_end_adr, Node* size_in_bytes) {
  if (UseTLAB) {
    if (UseZGC) {
        // get different TLAB based on size
    } else {
        // original TLAB code
    }
  } else {
      // original Eden code
  }
}
```

Listing 3.13: Structure of Modified C2 Allocation code

We proposed two ways of getting the corresponding TLAB based on the object size, using the two different types of conditional node provided in `C2` Compiler: the Conditional Move node and the IF node.

Using the Conditional Move Node

To use a Conditional Move Node, a `BoolNode` and a `CmpLNode` (Compare Long Integer Node) should be generated first to control the results. The code shown in Listing 3.14 could generate a `BoolNode` by comparing the size and size limit(`transform_later`s are omitted). The `BoolTest::le` means "less or equal", so the `BoolNode` will be `true` when the size is smaller or equal to the tiny object threshold.

```
Node* istiny_cmp = new CmpLNode(size_in_bytes, MakeConX(
    ZObjectSizeLimitTiny));

Node *istiny_bol = new BoolNode(istiny_cmp, BoolTest::le);
```

Listing 3.14: Getting a BoolNode in C2

Then, two sets of `top` and `end` offsets are generated to work as a candidate for final results, as shown in Listing 3.15. The resulting offset is created by constructing a Conditional Move Long Integer Node `CMoveLNode` with the `BoolNode`, two candidate offsets, and the type value, as shown in Listing 3.16. The final results are generated by the same `basic_plus_adr` function.

```
Node* tlab_small_top_offset = MakeConX(in_bytes(JavaThread::
    tlab_top_offset()));
Node* tlab_small_end_offset = MakeConX(in_bytes(JavaThread::
    tlab_end_offset()));
Node* tlab_tiny_top_offset = MakeConX(in_bytes(JavaThread::
    tlab_tiny_top_offset()));
Node* tlab_tiny_end_offset = MakeConX(in_bytes(JavaThread::
    tlab_tiny_end_offset()));
```

Listing 3.15: Get Top and End Offsets in C2

```
Node *tlab_top_offset = new CMoveLNode(istiny_bol,
    tlab_small_top_offset, tlab_tiny_top_offset, TypeLong::
    LONG);

Node *tlab_end_offset = new CMoveLNode(istiny_bol,
    tlab_small_end_offset, tlab_tiny_end_offset, TypeLong::
    LONG);
```

Listing 3.16: Get offset result in C2

```
eden_top_adr = basic_plus_adr(top(), thread, tlab_top_offset)
    ;
eden_end_adr = basic_plus_adr(top(), thread, tlab_end_offset)
    ;
```

Listing 3.17: Get the final results in C2

Using the IF Node

The Conditional Move Node is easy to construct but is not able to be optimized by the `C2` Compiler. We can maximize the optimization and performance by using an `IFNode`. With the `IFNode`, the `C2` Compiler can prune the tree to merge some branches or even make branch predictions by the arguments provided to the `IFNode`.

A basic branch structure can be created by the code shown in Listings 3.18. The `IFNode` should be contained in a `Region` with three fields. Also, it needs a `Control` input to be functional. The `Control` is the key point of `C2`'s optimization. `C2` will follow the `Control` to remove all unneeded nodes and merge branches. It is also the difficult part in `C2` development since we need to receive the `Control` from upstream and pass the `Control`s to downstream. For the `IFNode`, it creates two `Control` signals: `IFTrueNode` and `IFFalseNode`. Both of them should be correctly connected to another node, or the compilation will fail.

```
Node *istiny_reg = new RegionNode(3);
Node* istiny_cmp = new CmpLNode(size_in_bytes, MakeConX(
    ZObjectSizeLimitTiny));
transform_later(istiny_cmp);
Node *istiny_bol = new BoolNode(istiny_cmp, BoolTest::le);
transform_later(istiny_bol);
IfNode *istiny_if = new IfNode(ctrl, istiny_bol, PROB_LIKELY
    (0.9), COUNT_UNKNOWN);
transform_later(istiny_if);
Node *istiny_ift = transform_later(new IfTrueNode(istiny_if))
    ;
Node *istiny_iff = transform_later(new IfFalseNode(istiny_if)
    );
istiny_reg->init_req(1, istiny_ift);
istiny_reg->init_req(2, istiny_iff);
```

Listing 3.18: Basic branch code block in C2

A `PhiNode` is used to collect results from the branch. The `PhiNode` needs to be created within the `Region` and bind corresponding input nodes to the results. The code shown in Listings 3.19 creates the `PhiNode` with the previous `Region` and binds `eden_tiny_end_adr` to `IFTrueNode` and `eden_end_adr` to `IFFalseNode`. So the actual value of this `PhiNode` will be changed based on the result of the `IFNode`.

```
Node* phi = new PhiNode(istiny_reg, Type::BOTTOM);
phi->init_req(1, eden_tiny_end_adr);
phi->init_req(2, eden_end_adr);
```

Listing 3.19: PhiNode in C2

It is hard to judge the performance impact of the nodes introduced in both `CmpLNode` and `IFNode` methods. Unlike the `C1` Compiler, arrays and other objects share the same allocation path in `C2`. It appears to be that `C2` is not optimized because, for normal objects, no extra native code was generated in `C1`, but an extra node was generated in `C2`. However, `C2` could prune these tree nodes automatically, so it is expected that the added node will be removed (or precompiled) since the size of normal objects is fixed. There is also a possibility of bringing the same optimization to arrays. Futhermore, the allocation grammar tree in `C2` contains thousands of `Node`s, so it is unreliable to theoretically analyze the impact of this extra `Node`.

3.5 The Implementation of Relocation

The pseudo-code for relocate objects in HCSGC is shown in Listings 3.20. As can be seen in the code, when an object is to be relocated, its address will be searched in the per-page forwarding table. The forwarding table, which is provided by ZGC, is used to record a map from old addresses to the new one when accessing stale pointers. If the forwarding table contains that address, the object will be fully relocated by ZGC. In ZGC, if the object is not on the forwarding table, it will not be relocated. However, in HCSGC with partial evacuation enabled, the whole page will be selected to relocate. In this case, they will not be on the forwarding tables and be relocated by a separate function `relocate_object_in_pec`. In that function, the size of object will be calculated before moving the object to the new location.

```
uintptr_t ZHeap::relocate_object(uintptr_t addr) {
  ZForwarding* forwarding = _forwarding_table.get(addr);
  if (forwarding == NULL) {
    if (UsePartialEvacuation) {
      page = _page_table.get(addr);
      if (page->is_relocatable() && page->type() ==
   ZPageTypeSmall) {
        // HCSGC Partial Evacuation Relocation
        return relocate_object_in_pec(page, addr);
      }
    }
    // Not forwarding
    return ZAddress::good(addr);
  }

  // ZGC Full Relocation
  return relocate_object(forwarding, addr);
}
```

Listing 3.20: Pseudo code of object relocation in HCSGC

We proposed two methods of checking whether the object needs to be relocated, by the page type and by the object size. The page type check can be implemented by simply changing the `ZPageTypeSmall` to `ZPageTypeTiny`, as shown in Listings 3.21. The object size check is shown in Listings 3.22. To optimize this, size of the object is passed to the `relocate_object_in_pec` function to prevent double size calculation of the same object.

As proposed in Section 3.2.2, the object size calculation is much slower than the page type check. The code shown in Listings 3.21 can bail out early if the object is not on the tiny page and prevent the size calculation, while the other code always needs to calculate the size. Therefore, we could suggest that the page type check will be faster.

```
page = _page_table.get(addr);
if (page->is_relocatable() && page->type() == ZPageTypeTiny)
   {
    // HCSGC Partial Evacuation Relocation
    return relocate_object_in_pec(page, addr);
}
```

Listing 3.21: Pseudo code of checking by page type

```
size_t size = ZUtils::object_size(addr);
if (size > ZObjectSizeLimitTiny) {
    // Not forwarding
    return ZAddress::good(addr);
}
page = _page_table.get(addr);
if (page->is_relocatable()) {
    // HCSGC Partial Evacuation Relocation
    return relocate_object_in_pec(page, addr, size);
}
```

Listing 3.22: Pseudo code of checking by object size

Chapter 4

Evaluation Methodology

Relocating fewer objects could improve the relocation performance, but with a locality trade-off. We proposed in the previous chapter that larger objects will not benefit from moving them around. Thus, ignoring these objects in relocation could avoid the impact on overall performance and at the same time reduce the size of the relocation set. A smaller relocation set leads to a higher relocation performance, which may enhance the overall throughput in constrained environments. There are several aspects we will discuss in this chapter:

- **Ways to measure relocation performance.** The throughput of a program can be indicated by the total time to finish the program or the count of operations finished in a specified period. However, the relocation performance could be difficult to tell. Theoretically, the relocation performance is inversely proportional to the size of the relocation set. We can get a numerical measurement of relocation performance by recording the relocation set size of each GC and do the statistics. Practically, we want something that could improve the system to some extent, especially some speed up.
- **The performance of size identification method.** Two methods of checking the size of objects are proposed in Section 3.2.1. We suggest that the tiny page could be more efficient since it can skip a time-consuming step in some cases. However, the actual performance difference introduced by the two methods is still unknown and could change in different benchmarks. Besides,

the page division introduced an extra check when objects are being allocated, we want to know the overhead introduced by the page division.

- **The optimal size limit for tiny objects.** As discussed in Section 3.1, the size limit is a balance between the locality and the relocation performance, and this balance differs between environments and applications.

  In constrained systems, a smaller relocation amount means a smaller workload, and a lower memory fragmentation means a lower memory footprint and less GCs, which both lead to better performance. A small size limit may cause poor locality, but the improvements in relocation will compensate for this. Therefore, the size limit could be reduced drastically.

  In over-provisioned systems, we have enough memory to deal with the fragmentation and enough CPUs to deal with the high relocation workload. A lower relocation performance will not affect the throughput by any means, but the locality does. Therefore, only the large object that could never get any locality benefits can be ignored. We need to avoid introducing visible degradation by the page division, so the size limit needs to be higher than constrained environments.

  We introduced a command-line parameter to set the size limit in Section 3.3. End-users could tweak this value to get the best performance for their programs. Otherwise, a default value for the size limit is needed.

- **The selection of benchmarks.** According to the nature of tiny pages, it could only get performance improvements when there are an observable amount of objects that are larger than the limit. However, in most of the programs, the vast majority of objects are very small. In these cases, the tiny page would not make any difference from the original HCSGC since they are relocating the same amount of objects. These benchmarks could prove whether the page division will introduce performance degradation. Nevertheless, it is more meaningful to find benchmarks that could benefit from our proposal.

4.1 Measuring Method

A reasonable way to measure the relocation performance is critical in evaluation. We did this from two aspects. First, we record the relocation status of each GC. Less GC count and smaller relocation size is the sign of higher relocation performance. Second, we measure the overall throughput of the benchmark. If the result is better than HCSGC, we can infer that we got a larger relocation performance improvement with a smaller loss in the locality.

Recording the GC status, especially the relocation size, will introduce degradation to the overall performance. Thus, the overall throughput should be benchmarked without extra debug flags or GC statistics. As a result, these two aspects must be measured in different benchmark runs.

4.1.1 Record GC status

To get a clear outline of GC status, we collect data from several aspects, including the percentage of tiny/small objects, the count of GC cycles, and the total size and count of relocations. We also recorded the execution time as a reference.

- The percentage of objects with different size could be extracted from the built-in GC statistics in JRE. The command-line parameter `-Xlog:gc+reloc` could print logs that have both `gc` and `reloc` (means relocation) tags. A typical output of the relocation statistics is shown below.

```
[gc,reloc] GC(1) Tiny Pages: 25 / 50M(20%), Empty: 0M(0%),
            Compacting: 0M(0%)->0M(0%)
[gc,reloc] GC(1) Small Pages: 71 / 142M(55%), Empty: 0M(0%),
            Compacting: 132M(52%)->72M(28%)
[gc,reloc] GC(1) Medium Pages: 1 / 32M(12%), Empty: 0M(0%),
            Compacting: 0M(0%)->0M(0%)
[gc,reloc] GC(1) Large Pages: 4 / 32M(12%), Empty: 6M(2%),
            Compacting: 0M(0%)->0M(0%)
[gc,reloc] GC(1) Relocation: Successful
```

  From the log, we know that in this particular GC cycle, the tiny objects take 20% of the heap space, and the small objects take 55% of the heap space. A single benchmark run often contains

hundreds of GC cycles. We calculate the overall percentage of tiny objects by averaging all GC cycles.

- The count of GC cycles could be extracted by another parameter, `-Xlog:gc`. The following line will be printed after each GC, where *N* is the current count of GC cycles and *Allocation Rate* is the reason why this GC happens.

```
[gc      ] GC(N) Garbage Collection (Allocation Rate)
```

  We could get the total GC cycles by finding the maximum *N* in the log.

- The relocation size counter is not provided by default. We could create one by the `ZStatCounter` constructor. The related code is shown in Listing 4.1. `ZStatUnitBytesPerSecond` indicates that the unit of the counter is bytes per second, which is used to count the relocation size. While the `ZStatUnitOpsPerSecond` means the unit is operations per second, and we use it to count the number of objects that are being relocated. The counter could be increased by the `ZStatInc` function.

```
static const ZStatCounter ZCounterRelocationSize("
    relocationsize", "Relocation Size",
    ZStatUnitBytesPerSecond);
static const ZStatCounter ZCounterRelocationCount("
    relocationcount", "Relocation Count",
    ZStatUnitOpsPerSecond);
```

Listing 4.1: ZStatCounter Creation

As described in Section 2.4, the HCSGC defers the relocation phase of some pages (small pages in HCSGC, or tiny pages in our implementation) to the next GC cycle. It means we have two types of relocations in HCSGC, the full relocation and the lazy relocation. The full relocation may happen in both mutator threads and the concurrent GC threads, while the lazy relocation could only happen in mutator threads. To deal with this, we add the size and operations counter for all combinations of different relocation types and different threads.

The statistics could be printed by the command-line parameter `-Xlog:gc+stats`. It can be combined with the previous

ones and become `-Xlog:gc,gc+reloc,gc+stats`, or simply output all GC-related logs by `-Xlog:gc*`.

4.1.2 Measure overall throughput

The throughput difference could be small, so it is critical to make the results more reliable. We designed the layered benchmark flow (Figure 4.1) to narrow the confidence interval.

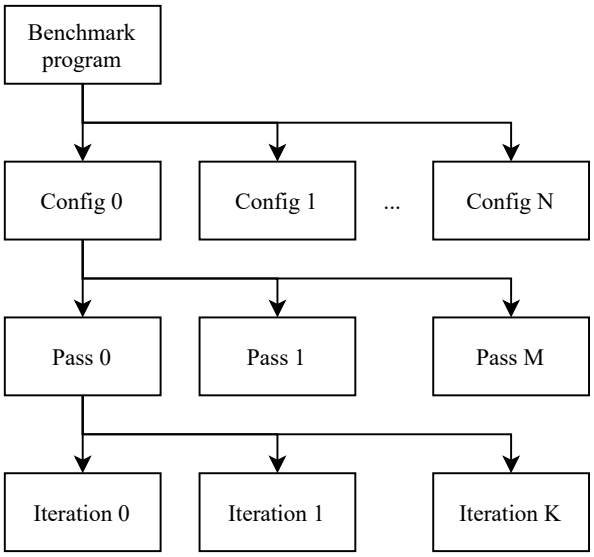


Figure 4.1: The benchmark flow

For each of the benchmark programs, several different configurations will be applied, which contains several passes. Every pass is an invocation to the benchmark program, which will generate the time data such as wall-clock time. This data will be collected and calculate the average, standard deviation, and confidence interval. There are usually 5 to 30 passes for each configuration, depending on the stability of the results.

Iteration is the internal loop of each benchmark. Usually, the benchmark program will provide a command-line argument to set the number of iterations. The iteration time will be printed after each iteration.

The reason of using iterations is the *Tiered Compilation* mentioned in Section 2.1.1. The first several iterations are usually executed by

the C1 Compiler for a faster startup. The C2 Compiler will work after the third or fourth iteration and generate a faster assembly. Usually, the benchmark gets stable after this. But the C2 will collect runtime information and try to generate further optimized code. Therefore, some of the benchmarks may be unstable even after ten iterations. When the time gets stable, we make it run ten more iterations and collect the average of the last ten stable iterations as the final result for this pass.

If we remove the iterations (so there is only one iteration in each pass), the code will always run in the less optimized mode and gets a lower throughput.

The pseudo-code of the iterations is shown in Listing 4.2. Notice that we put a force GC `System.gc()` at the end of each iteration but without timing on it. It could assure the memory layout is the same at the beginning of each GC to prevent unstable values.

```
for (int i = 0; i < iterations; i ++) {
    var start = System.currentTimeMillis();
    // <workload>
    var end = System.currentTimeMillis();
    System.out.println("Benchmark X Iteration " + i + "
   PASSED in " + (end-start));
    System.gc();
}
```

Listing 4.2: Iterations - the internal loop

4.1.3 Common configurations

Following common parameters are used for all of the benchmarks.

- The default HCSGC switch `-XX:+UsePartialEvacuation` and `-XX:+UseLazyRelocate` will be enabled for all benchmarks except the vanilla ZGC.
- Constrained environments need to be created for most of the benchmarks to get throughput improvements. Memory-constrained environment can be created by limiting the Java heap size, with the JVM arguments `-Xmx<size>` and `-Xms<size>`.
- Creation of CPU-constrained environments differs according to the benchmark program. For single-threaded benchmarks,

Table 4.1: Configuration used in benchmarking

Configuration No.	0	1	2	3	4	5	6	7	8	9
Garbage Collector	HCSGC	Tiny page								Vanilla
ZObjectSizeLimitTiny	n/a	262144	512	256	255	192	128	64	32	n/a

we use the `taskset -c 0 <workload>` command to force both mutator and GC thread to run on the same core so they will compete for the limited resource. For multi-threaded ones, the benchmark program itself will exhaust the hardware. Therefore, no extra operations need to be taken.

4.2 Tiny page threshold

We run the benchmark with different thresholds for tiny objects to test for the best size limit of each benchmark. As shown in Table 4.1, different size limits from 32 bytes to 256K bytes are chosen, as well as the original HCSGC and vanilla ZGC. The threshold is set to the corresponding number by the command-line parameter `-XX:ZObjectSizeLimitTiny=<size>` proposed in Section 3.3.

Config 0 is baseline benchmark runs with the original HCSGC. Config 1, which has a 256K bytes limit, should have no small pages, and all previous small objects in HCSGC would now be on tiny pages and be lazy relocated. Ideally, Config 1 should have the same performance as Config 0. Comparing the results of Config 1 to Config 0 could show the overhead introduced by the page division. Config 9 runs with vanilla ZGC as a reference.

With the size limit decreases from left to right, the locality will also decrease. For the benchmarks run in unconstrained environments, the relocation performance would not affect the throughput. As a result, the overall performance decreases as the locality decreases. If we find such a number N that Config N has a similar performance as Config 0, while Config N+1 has a degradation, then Config N is the optimal threshold for the current benchmarking program in unconstrained environments.

For the constrained environments, if we got an overall improvement, that is likely come from the relocation. Therefore, we can easily get the threshold with the best performance by finding the shortest

execution time.

Note that we added a special threshold of 255 into the configurations. The performance difference between 255 Bytes and 256 Bytes threshold is only introduced by whether ignoring the 256 Bytes objects. This could identify the relocation improvement and locality loss of objects that are exactly 256 Bytes. We select this value based on the following reasons.

- 256 Bytes suits the "slightly larger than cache line size" that we want. As described in Section 3.1, objects smaller than the size of the cache line will get benefits from the locality. Objects that are slightly larger than that have a chance of getting benefits, but we also need to pay the price of reducing relocation performance. Considering the CPUs have a maximum of 13 or 20 extra prefetches, 256 Bytes, which equals four times of cache line size and needs 3 or 4 prefetches to load the whole object, is ideal for our "slightly larger" size.
- Several benchmark programs have a considerable amount of objects that are exactly 256 Bytes, making it easier for us to discover any performance difference.

End-users could tweak this threshold to get the best performance for their programs. Otherwise, a default value for the size limit is needed. We set the default threshold to the value that would not introduce any degradation on most of (if not all) the benchmarks in unconstrained environments. That value is selected because: (1) this is the smallest threshold that would not lower the performance comparing to original HCSGC, (2) this is the best improvement over relocation performance that we can get (with the restriction of (1)).

4.3 Benchmark software

4.3.1 Synthetic benchmark

The synthetic benchmark is used as a form of sanity check, that exhibits a stable but unpredictable access pattern. The benchmark illustrates the HCSGC could capture the pattern and reorganize the objects to improve cache utilization. [8]

In the original benchmark, an array of 2×10^6 `Pair`s is created. Each `Pair` contains two numbers. A loop will persist on accessing the array, and at the same time, allocating some garbage to trigger GC. From the page's perspective, the synthetic benchmark has one large page, which stores the large 2×10^6 array. All other objects, including the temporary variables when doing the workload, is very small and will live on tiny pages. So it would not get any difference in performance between HCSGC and tiny pages.

To illustrate the usage of tiny pages, we changed the synthetic benchmark a bit by extending the original `Pair` class to a larger `LargePair` class, which contains 1000 numbers. Then, we replaced part of the `Pair`s in the array with `LargePair`s. As a result, we can get some small pages. The proportion of replaced objects could be changed to change the overall small page percentage. In this benchmark, we changed 5% of `Pair`s into `LargePair`s and get 80% of small pages. With tiny page enable, this 80% of objects could be ignored from relocation, thus we can suppose a performance improvement.

4.3.2 DaCapo Suite

The DaCapo benchmark suite is a collection of carefully selected open-source real-world applications with complex logic. All benchmarks in the suite are considered to offer a unique aspect of the performance of JVM [24]. The latest stable release of DaCapo is DaCapo-9.12-bach-MR1, which was released in 2009. Although old, it is still used by the community as it provides credibility. For this thesis, a subset of the DaCapo will be used. We use the largest input size for each benchmark we choose. The detail of selected benchmarks and input size is shown in Table 4.2. For the other benchmarks that are not chosen, the reasons are as follows.

- Some of the benchmarks could not even start. The DaCapo developed a long time ago, with a quite old version of JDK. Many benchmarks have compatibility issues with newer versions such as the JDK 15 which is used by this thesis.
- Some of the benchmarks lack larger objects. We will not run benchmarks that have more than 99% of tiny pages with the default threshold (256 Bytes). The difference in

Table 4.2: Brief description of selected benchmarks in DaCapo

benchmark	size	description
avrora	large	Simulates a number of programs run on a grid of AVR microcontrollers.
fop	default	Takes an XSL-FO file, parses it and formats it, generating a postscript file.
h2	huge	H2 is an SQL relational database engine written in Java. This benchmark executes a TPC-C like in-memory benchmark, executing the following models: customers, districts, warehouses, purchases and deliveries.
luindex	default	Indexes a set of documents, the works of Shakespeare and the King James Bible.
lusearch	large	Text search of keywords over a corpus of data comprising the works of Shakespeare and the King James bible.
sunflow	large	Renders a set of images using ray tracing.
xalan	large	Xalan is an XSLT processor for transforming XML documents into HTML. This benchmark will repeatedly transforms a set of XML documents.

relocation amount will be indistinguishable if most of the previous small objects transfer to tiny objects and be relocated, and the relocation performance improvement could be even imperceptible. We could lower the tiny object threshold to put more objects on small pages and ignore more objects in relocation, but this would definitely lower the overall performance since these objects could benefit from the locality.

4.3.3 JGraphT

The JGraphT is a Java library of graph theory data structures and algorithms, focusing on flexibility, powerfulness, and efficiency [25]. We run two algorithms from the JGraphT as benchmarks. The first is the Bron-Kerbosch maximal clique enumeration algorithm, *maximal clique (MC)* [26]. The other is a weakly biconnected components algorithm, *connected components (CC)* [27]. We use

the `uk-2007-05@100000` graph data from Laboratory for Web Algorithms (LAW) [28, 29].

HCSGC implemented a minimal driver that only loads the graph and calls a method from JGraphT on the graph to work as a benchmark [30]. To keep a consistent result, we will use that implementation for the JGraphT benchmark.

4.3.4 SPECjbb2015

The SPECjbb® 2015 benchmark has been developed from the ground up to measure performance based on the latest Java application features. It is relevant to all audiences who are interested in Java server performance. [31]

A typical SPECjbb2015 benchmark run takes about 2 hours, which contains several phases shown in Figure 4.2. The benchmark starts with the Search HBIR phase. High Bound Injection Rate (HBIR) is an estimation of the maximum injection rate that the system can handle. In the later phase, Response-Throughput (RT) curve building, the RT step value is increased by 1% of HBIR. The HBIR is calculated by running different injection rate levels for a fixed duration to determine whether the system is able to successfully execute the level. It is an estimation of throughput without any latency constraints.

The benchmark has two metrics, `max-jOPS` and `critical-jOPS`. The `max-jOPS` indicates the throughput of the system, which is calculated by the last success injection rate of RT step level before the first failure of an RT step level. The `critical-jOPS` indicates the latency of the system, which is the geo-mean of jOPS in 10ms, 25ms, 50ms, 75ms, and 100ms response time. The jOPS is measured by the 99^{th} percentile response for all requests.

Although the `HBIR` and `max-jOPS` both measure the throughput, they have different values and the `max-jOPS` usually between 70%-90% of `HBIR`. A much lower `max-jOPS` than `HBIR` may indicate the system has occasional long pauses [32].

There is an option to execute the benchmark on more than one machine to benchmark on network I/O. Since the garbage collector is not affected by the network, we run SPECjbb2015 with the composite setting (single VM, single host). The heap size is set to 64GB.

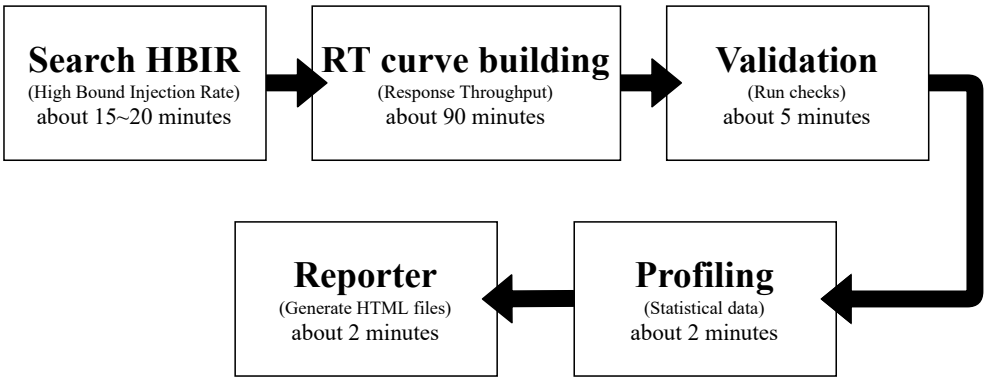


Figure 4.2: The SPECjbb2015 run progress

4.4 Machines to Collect Data

As shown in Table 4.3, we have two machines to run the benchmark. The larger memory machine A has an AMD Ryzen 9 3900X @ 3.8GHz with 12 cores (2 hyperthreads/core), 96GB RAM, 64KB L1(per core), 512KB L2(per core), 64MB L3(by 4x16MB), and running Ubuntu 20.04.2.0 LTS (Focal Fossa) with Linux kernel version 5.8.0-53-generic. The other machine B has an Intel® Core™ i7-4710mq CPU @ 2.5GHz with 4 cores (2 hyperthreads/core), 8GB RAM, 64KB L1(per core), 256KB L2(per core), 6MB L3(shared), also running Ubuntu 20.04.2.0 LTS (Focal Fossa) with Linux kernel version 5.8.0-53-generic.

The SPECjbb2015 benchmark requires a large heap, so it runs on machine A. Other benchmarks runs on machine B.

The C/C++ compiler used is GCC 7.5.0, and the OpenJDK version we based on is JDK 15.

4.5 Evaluation Design

We aim to evaluate the following aspects of our implementation of tiny pages.

1. **The better size identification method.** Two methods of checking the size of objects are proposed in Section 3.2.2, named the naive method, which checks the size of each object first, and the tiny page method, which checks the page type first. We also provided an alternative method that avoids the overheads

Table 4.3: Machines to Collect Data

Configuration	Machine A	Machine B
CPU	AMD Ryzen 9 3900X @ 3.8GHz with 12 cores (2 hyperthreads/core)	Intel® Core™ i7-4710mq CPU @ 2.5GHz with 4 cores (2 hyperthreads/core)
Memory	96 GB DDR4@2400MHz	8 GB DDR3@1600MHz
L1 Cache	64KB (per core)	
L2 Cache	512KB (per core)	256KB (per core)
L3 Cache	64MB	6MB
System	Ubuntu20.04.2.0 LTS (Focal Fossa)	
Kernel	Linux kernel version 5.8.0-53-generic	

in allocation but discards the fragmentation improvements in Section 3.2.3. Theoretically, we suggest that the tiny page method could be more efficient since it can skip a time-consuming step in some cases. We also state that the alternative way does not benefit since fragmentation plays a critical role in performance. We need to validate these proposals by benchmarks. After finding the best size identification method, we could settle the implementation for subsequent benchmarks.

2. **The overhead of tiny pages.** As discussed in Section 3.4, there is a small but inevitable allocation overhead of using tiny pages since we need to check whether the object is tiny or small. We want to know the degradation introduced by this overhead. Besides, we proposed two types of size checking in `C2` Compiler in Section 3.4.3, the `CMoveLNode` and the `IFNode`. We want to know which one is better. These could be detected by comparing the overall throughput difference of Config 0 and 1 shown in Table 4.1.

3. **The optimal size limit for tiny objects.** As discussed in Section 4.2, we need to find the optimal threshold that would not introduce degradation on most of the benchmarks in unconstrained environments.

4. **The throughput improvements or regressions.** The improvements in relocation may reflect in the overall throughput. We want to know what is the best throughput improvements we can get in each benchmark program. Also, we want to check whether any benchmark gets regression after introducing our proposal.

5. **Analysis of relocation performance improvements from the GC status view.** The relocation performance is related to the memory fragmentation and the relocation count/size. We want to know these values in different tiny page thresholds to find relations between relocation performance and overall throughput.

Chapter 5

Results and Discussion

5.1 Comparison of size identifying methods

We proposed the following three implementations of reducing the relocation amount.

I1 Enable the tiny page. When relocating, check the page type and only relocate objects that are on tiny pages.

I2 Enable the tiny page. When relocating, check the object size and only relocate objects that are smaller than the tiny threshold.

I3 Disable the tiny page. When relocating, check the object size and only relocate objects that are smaller than the tiny threshold.

We try to find the best method by comparing the throughput with different tiny object thresholds and environments. Six configurations are selected to test this, as shown in Table 5.1. The selection is based on the following reasons.

- To maximize the difference, we use benchmarks that gets improvements in HCSGC, so the H2 and JGraphT benchmark are selected.
- Implementation **I3** will keep the memory fragmented, so it is supposed to have lower performance when the heap size is small. Therefore, we will evaluate these in both memory-constrained and -unconstrained environments.

Table 5.1: Benchmark configuration for comparison of size identifying methods

Benchmark	Heap size
DaCapo H2	Unconstrained: 6GB Constrained: 3GB
JGraphT connected_components	Unconstrained: 1GB Constrained: 512MB
JGraphT maximal_clique	Unconstrained: 1GB Constrained: 512MB

- The size check is handled during lazy relocation, which is processed in worker threads. It is competing with mutators regardless of the environment. Therefore, there is no need to evaluate in CPU-constrained environments.

The results of each benchmark are presented in Table 5.2, 5.3, 5.4, respectively, and visualized in Figure 5.1, 5.2, 5.3. The results are collected as wall-clock time in seconds, a lower result means a better throughput. In the table, `HCSGC` indicates the results are from original HCSGC, while the others following the format "$\texttt{I}_{N}\texttt{-}\mathit{Size}$" where N represents the Nth implementation presented above, and Size represents the tiny page threshold.

We see that Implementation **I1** has the same or better performance than **I2** in all benchmarks. The only difference between the two implementations is the tiny object identification method. The results indicate that the tiny page method has a better performance than the naive method.

However, the results of unconstrained environments may give the impression that Implementation **I3** has the best performance. It is reasonable since memory fragmentation will not cause regressions when memory is large enough, and tiny pages will introduce extra overhead without getting any benefits in this situation.

We also see the same results in the constrained configuration of the H2 benchmark, as shown in Figure 5.1c. It may be caused by the following reasons. First, 3G Bytes of the heap is not constrained enough. From the throughput column of Table 5.2a and Table 5.2b, we can see that all configurations have similar performance, while other benchmarks got a 20%-30% regression in constrained

Table 5.2: Comparison of size identifying methods in DaCapo H2

(a) Unconstrained: 6G Bytes Heap

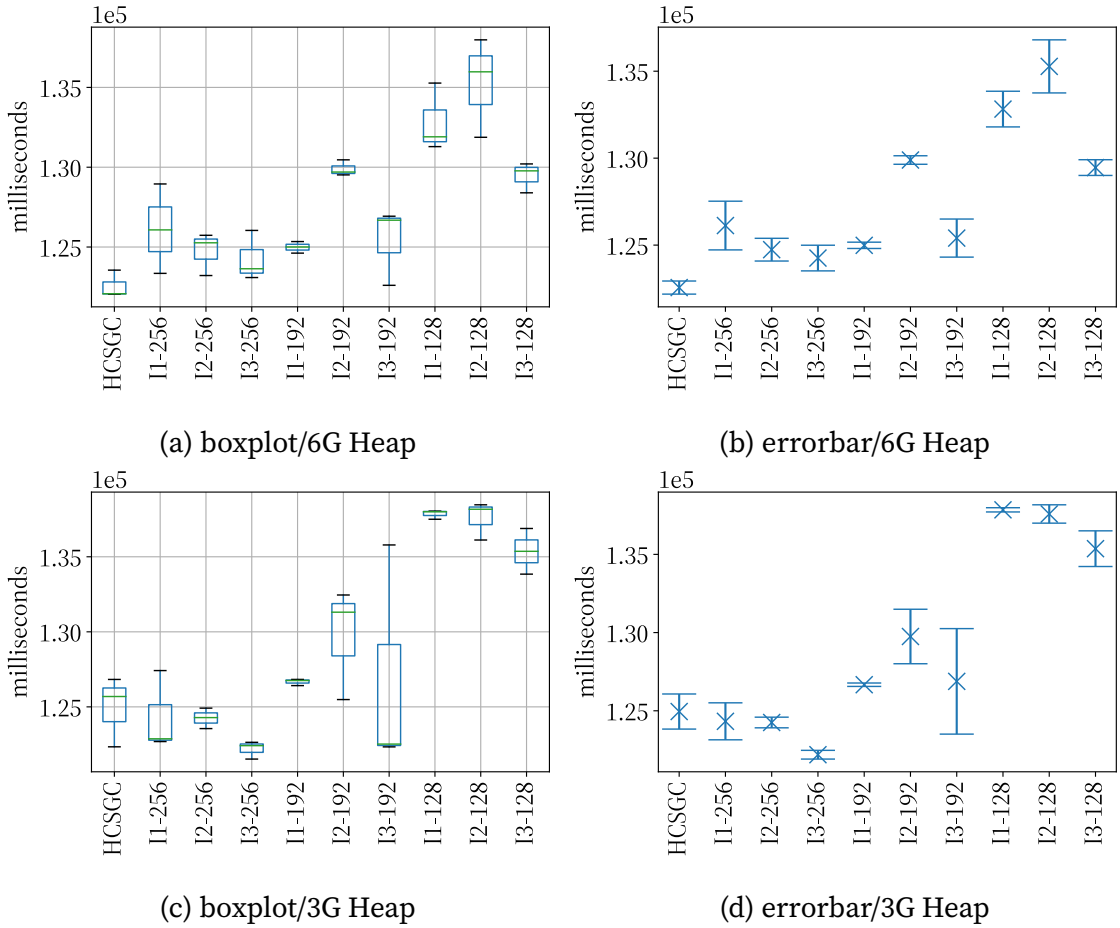
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	122558.00	860.39	0.70%	0.00%	122204.26	122957.78
I1-256	10	126124.47	2801.05	2.22%	-2.91%	124726.68	127527.38
I2-256	10	124737.47	1340.06	1.07%	-1.78%	124081.16	125358.44
I3-256	10	124255.33	1569.57	1.26%	-1.38%	123548.22	125024.12
I1-192	10	124987.33	362.63	0.29%	-1.98%	124805.38	125167.78
I2-192	10	129892.87	501.17	0.39%	-5.98%	129648.70	130138.56
I3-192	10	125403.80	2431.01	1.94%	-2.32%	124257.54	126422.34
I1-128	10	132822.00	2142.70	1.61%	-8.37%	131843.54	133924.36
I2-128	10	135276.87	3111.64	2.30%	-10.38%	133715.84	136767.94
I3-128	10	129457.93	944.70	0.73%	-5.63%	128989.88	129894.58

(b) Constrained: 3G Bytes Heap

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	124952.07	2333.47	1.87%	0.00%	123793.06	126037.26
I1-256	10	124326.73	2679.04	2.15%	0.50%	123198.70	125581.78
I2-256	10	124251.47	681.73	0.55%	0.56%	123909.22	124590.32
I3-256	10	122191.87	590.57	0.48%	2.21%	121901.80	122459.50
I1-192	10	126666.93	219.88	0.17%	-1.37%	126552.12	126765.56
I2-192	10	129749.87	3735.91	2.88%	-3.84%	127930.18	131413.68
I3-192	10	126881.13	7714.48	6.08%	-1.54%	123717.04	130462.66
I1-128	10	137847.47	300.98	0.22%	-10.32%	137700.24	137979.50
I2-128	10	137582.73	1277.75	0.93%	-10.11%	136937.44	138139.98
I3-128	10	135364.50	2150.88	1.59%	-8.33%	134223.83	136450.86

environments. This indicates that 3G Bytes are unable to introduce enough memory pressure. We have tried smaller heap sizes such as 2GB, but it would fail immediately with an out-of-memory error. Second, the H2 benchmark does not have enough garbage to make memory fragmented. Memory fragmentation in HCSGC is caused by the lazy relocation, which keeps the garbage until the next GC, so the garbage will live one GC cycle longer than the original behavior. However, since H2 is a database program, most of the objects in H2 live for a long time. If the object lives more than hundreds of GC cycles, the extra cycle will not have a visible effect on the overall memory usage.

In conclusion, we could say that Implementation **I1** which uses the tiny page to reduce memory fragmentation and relocation amount is the best option for constrained environments.



(a) boxplot/6G Heap

(b) errorbar/6G Heap

(c) boxplot/3G Heap

(d) errorbar/3G Heap

Figure 5.1: Comparing size identifying methods in DaCapo H2

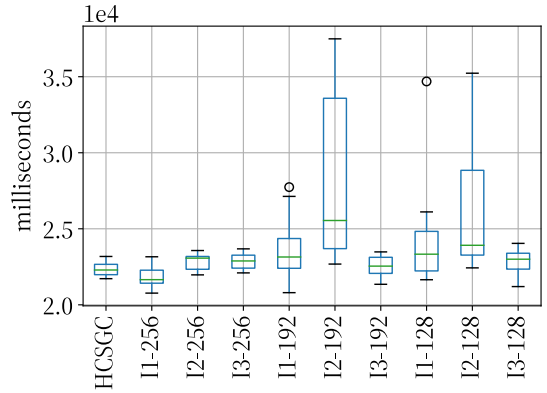
Table 5.3: Comparing size identifying methods in connected_components

(a) Unconstrained: 1G Bytes Heap

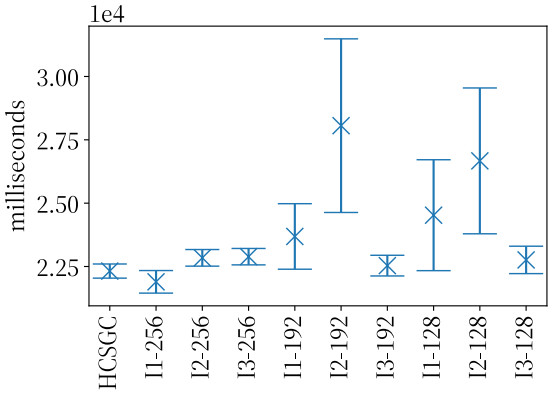
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	22320.00	474.72	2.13%	0.00%	22047.00	22609.00
I1-256	10	21895.00	768.19	3.51%	1.90%	21466.00	22362.03
I2-256	10	22842.00	564.44	2.47%	-2.34%	22488.00	23158.03
I3-256	10	22888.00	551.78	2.41%	-2.54%	22575.00	23213.02
I1-192	10	23686.00	2228.80	9.41%	-6.12%	22452.98	25068.00
I2-192	10	28059.00	5904.42	21.04%	-25.71%	24858.95	31664.32
I3-192	10	22535.00	701.11	3.11%	-0.96%	22130.00	22939.00
I1-128	10	24525.00	3845.74	15.68%	-9.88%	22761.92	27087.12
I2-128	10	26669.00	4957.87	18.59%	-19.48%	23971.98	29740.07
I3-128	10	22760.00	922.45	4.05%	-1.97%	22195.00	23279.00

(b) Constrained: 512M Bytes Heap

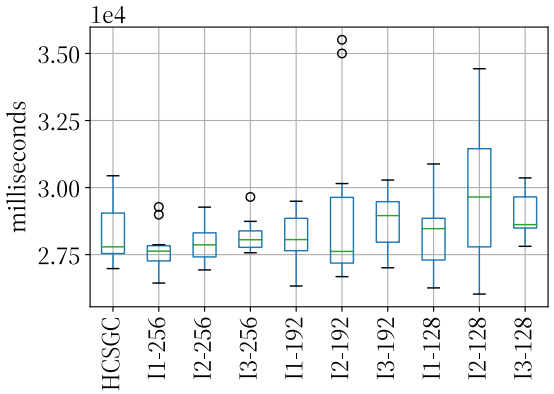
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	28300.00	1158.99	4.10%	0.00%	27615.00	29036.67
I1-256	10	27716.00	854.83	3.08%	2.06%	27243.00	28232.03
I2-256	10	27914.00	682.84	2.45%	1.36%	27535.00	28334.00
I3-256	10	28202.00	628.33	2.23%	0.35%	27878.00	28611.00
I1-192	10	28142.00	948.41	3.37%	0.56%	27563.00	28678.03
I2-192	10	29227.00	3317.70	11.35%	-3.28%	27462.98	31338.05
I3-192	10	28713.00	1061.89	3.70%	-1.46%	28099.98	29320.00
I1-128	10	28355.00	1394.14	4.92%	-0.19%	27547.97	29213.00
I2-128	10	29947.00	2810.63	9.39%	-5.82%	28338.98	31666.05
I3-128	10	28960.00	854.19	2.95%	-2.33%	28487.00	29474.03



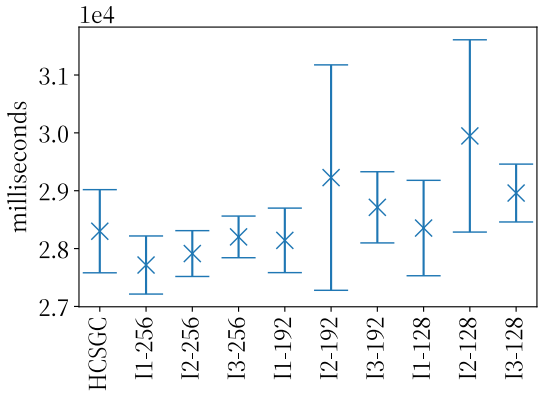
(a) boxplot/1G Heap



(b) errorbar/1G Heap



(c) boxplot/512M Heap



(d) errorbar/512M Heap

Figure 5.2: Comparing size identifying methods in connected_components

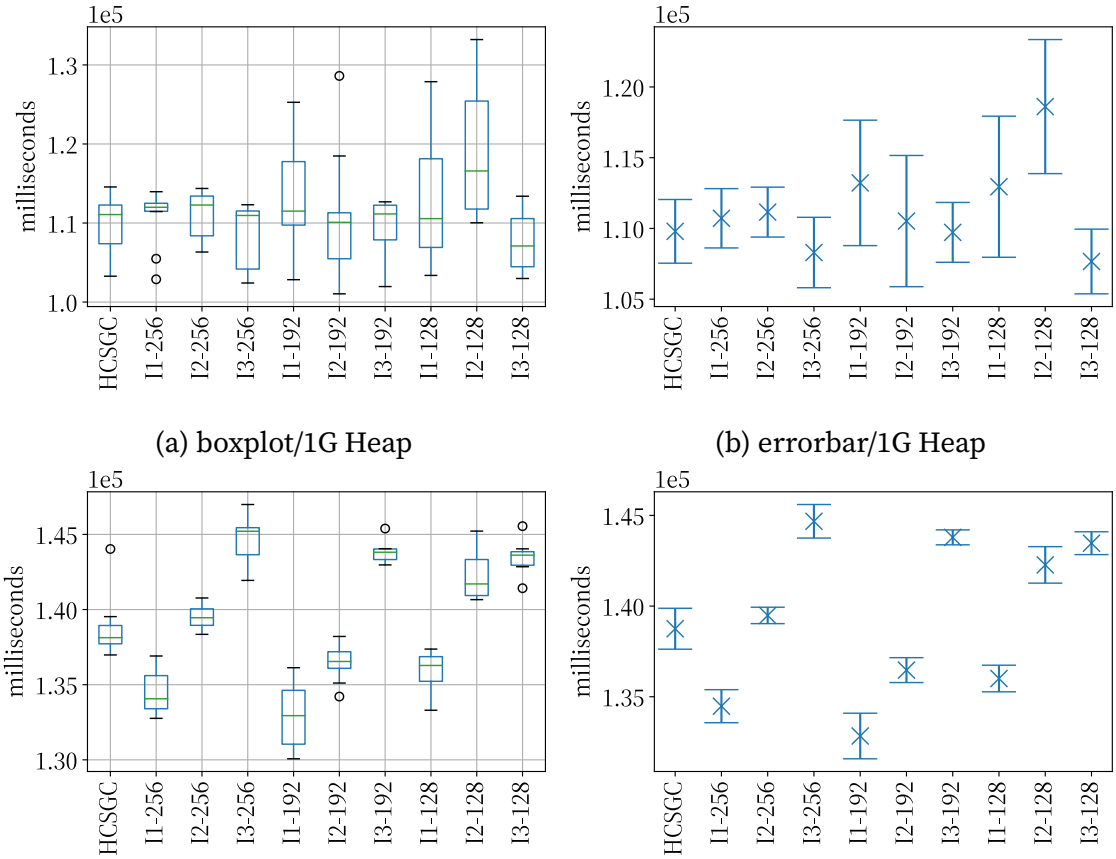
Table 5.4: Comparing size identifying methods in maximal_clique

(a) Unconstrained: 1G Bytes Heap

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	109794.00	3790.89	3.45%	0.00%	107423.00	111884.08
I1-256	10	110718.00	3574.42	3.23%	-0.84%	108379.97	112486.00
I2-256	10	111154.00	3032.50	2.73%	-1.24%	109291.98	112826.02
I3-256	10	108297.00	4199.98	3.88%	1.36%	105773.95	110720.02
I1-192	10	113219.00	7642.92	6.75%	-3.12%	108876.97	117769.10
I2-192	10	110520.00	8026.14	7.26%	-0.66%	106315.95	115785.20
I3-192	10	109722.00	3611.07	3.29%	0.07%	107442.90	111665.00
I1-128	10	112949.00	8676.44	7.68%	-2.87%	108172.00	118414.07
I2-128	10	118614.00	8234.70	6.94%	-8.03%	113967.85	123618.12
I3-128	10	107663.00	3907.20	3.63%	1.94%	105415.97	109944.00

(b) Constrained: 512M Bytes Heap

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	138750.00	2005.73	1.45%	0.00%	137813.00	140104.05
I1-256	10	134478.00	1542.86	1.15%	3.08%	133628.00	135414.02
I2-256	10	139484.00	781.26	0.56%	-0.53%	139029.00	139936.02
I3-256	10	144676.00	1579.40	1.09%	-4.27%	143719.95	145565.00
I1-192	10	132834.00	2186.84	1.65%	4.26%	131553.98	134119.00
I2-192	10	136468.00	1172.42	0.86%	1.64%	135750.00	137118.02
I3-192	10	143790.00	703.10	0.49%	-3.63%	143410.00	144225.00
I1-128	10	136004.00	1274.15	0.94%	1.98%	135171.98	136702.02
I2-128	10	142271.00	1701.56	1.20%	-2.54%	141327.00	143315.00
I3-128	10	143468.00	1059.54	0.74%	-3.40%	142840.00	144089.02



(a) boxplot/1G Heap

(b) errorbar/1G Heap

(c) boxplot/512M Heap

(d) errorbar/512M Heap

Figure 5.3: Comparing size identifying methods in maximal_clique

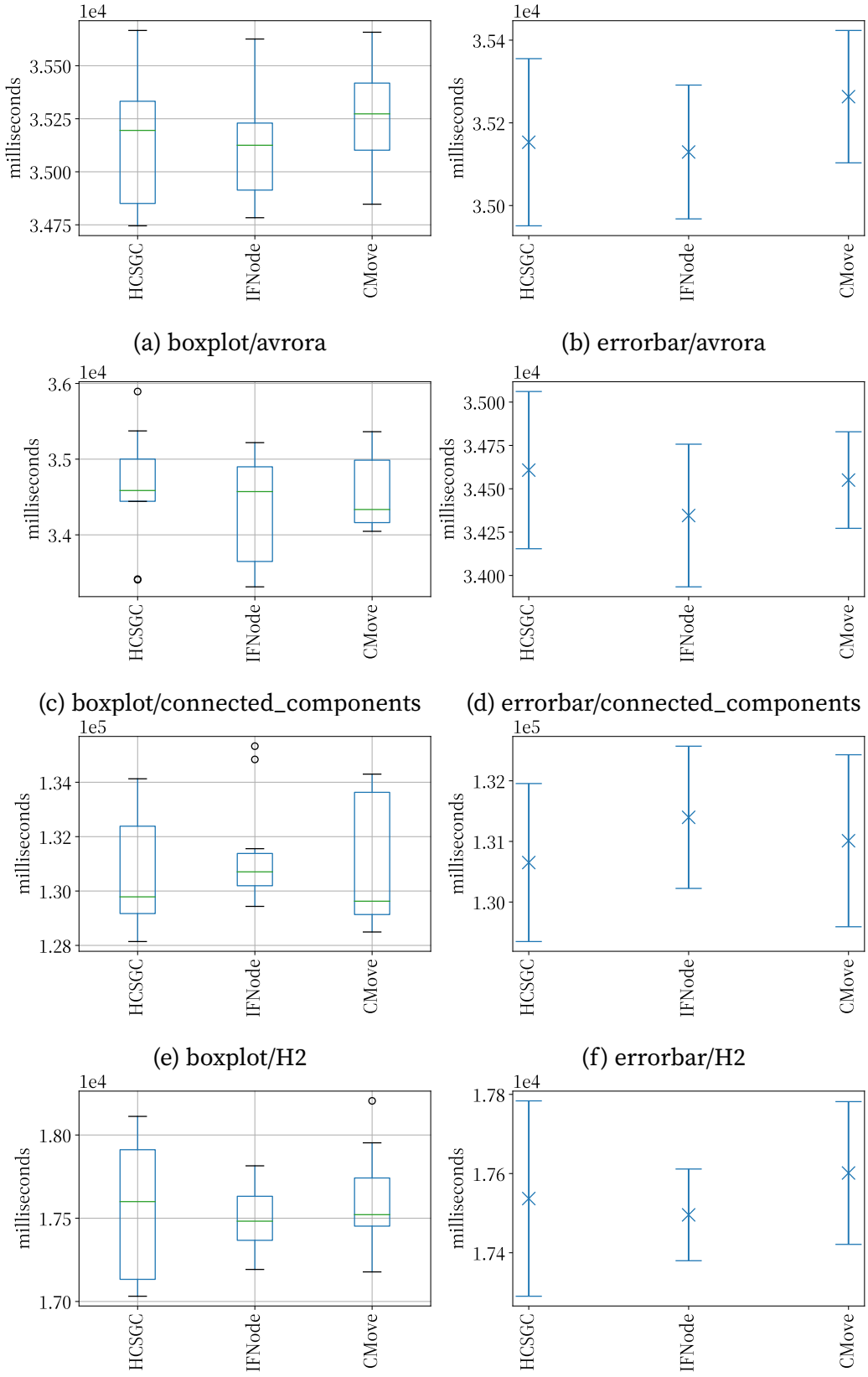
Table 5.5: Overhead of tiny pages

Benchmark	Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
avrora_large	CMove	10	35263.34	272.91	0.77%	-0.31%	35101.75	35422.31
avrora_large	HCSGC	10	35153.13	341.00	0.97%	0.00%	34957.13	35353.89
avrora_large	IFNode	10	35129.55	278.66	0.79%	0.07%	34974.08	35302.10
cc_uk	CMove	10	18723.75	428.10	2.29%	0.00%	18468.29	18969.10
cc_uk	HCSGC	10	18698.58	257.14	1.38%	0.13%	18554.17	18851.01
cc_uk	IFNode	10	18584.93	402.19	2.16%	0.74%	18339.57	18811.33
H2_huge	CMove	10	131011.01	2408.17	1.84%	-0.28%	129644.41	132474.18
H2_huge	HCSGC	10	130651.60	2216.83	1.70%	0.00%	129408.83	132047.64
H2_huge	IFNode	10	131397.90	2024.60	1.54%	-0.57%	130356.56	132682.00
xalan_large	CMove	10	17601.42	311.27	1.77%	-0.37%	17427.17	17790.95
xalan_large	HCSGC	10	17536.85	421.67	2.40%	0.00%	17287.87	17781.93
xalan_large	IFNode	10	17495.65	198.85	1.14%	0.23%	17380.82	17613.06

5.2 The overhead of tiny pages

In this part, we aim to measure the overhead of tiny pages. The avrora_large, H2_huge, xalan_large benchmark from DaCapo, and connected_components with UK dataset from JGraphT benchmark are selected. The results are collected as wall-clock time in seconds, presented in Table 5.5, and visualized in Figure 5.4. `HCSGC` indicates the results are from the original HCSGC, `CMove` is for the Conditional Move implementation, and `IFNode` is for the IF Node implementation. Both `CMove` and `IFNode` use 256K Bytes as the tiny page threshold to behave the same as the original `HCSGC`. Any regression in the results could represent the overhead introduced by the tiny page.

As shown in Figure figs. 5.4b, 5.4d, 5.4f and 5.4h, the confidence intervals of all three configurations are overlapping. Therefore, we could say the tiny page will not introduce visible performance degradation. However, we are unable to find a better implementation between `CMove` and `IFNode` since they have the same performance.



(a) boxplot/avrora

(b) errorbar/avrora

(c) boxplot/connected_components

(d) errorbar/connected_components

(e) boxplot/H2

(f) errorbar/H2

(g) boxplot/xalan

(h) errorbar/xalan

Figure 5.4: Overhead of tiny pages

5.3 Overall throughput benchmarks

In this part, we will go through all benchmarks with different configurations shown in Table 4.1 to find the best tiny page threshold for each benchmark. Then, the results will be analyzed and the default threshold will be selected. We will also check if there are any benchmarks getting regressions with our proposal.

5.3.1 JGraphT

We selected the `connected_components` and `maximal_clique` algorithm with the UK dataset to test the performance of the JGraphT library. Each configuration mentioned in Table 4.1 will be tested in unconstrained, memory-constrained, and CPU-constrained environments. The heap size of the memory-constrained environment is set to 512MB, while the other two are set to 1GB.

The results of `connected_components` are shown in Table 5.6 and visualized in Figure 5.5. In unconstrained environments, although all thresholds except `Tiny32` have overlapping confidence intervals with the HCSGC, we see the results get unstable and more than 1% regression starting from `Tiny255`. We could also find similar trends in the L1 cache miss rate, as shown in Figure A.1. We suggest that the benchmark has the possibility of having performance degradation with a 255B threshold. Therefore, the best value for `connected_components` is set to 256 Bytes.

The CPU-constrained configuration has a different behavior, where the HCSGC itself has a huge regression (-17.07%) comparing to the vanilla ZGC. It reveals that the relocation performance degradation introduced by HCSGC exceeds the locality it gained. As a result, less relocation leads to higher performance. We could find this from Figure 5.5f, where a lower threshold got a better throughput.

The results of `maximal_clique` are shown in Table 5.7 and visualized in Figure 5.6. Similar to the previous case, the benchmark gets overlapping results in most of the configurations. The results start to be unstable in the `Tiny255` config, so we set the best threshold for `maximal_clique` to 256 Bytes. In constrained environments, we see a maximum of 4.56% improvement with memory constraint and 3.19% improvement with CPU constraint.

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	22709.33	646.98	2.85%	0.00%	22319.00	23087.00
tiny256K	15	22776.00	505.18	2.22%	-0.29%	22478.97	23085.00
tiny256	15	22694.67	608.53	2.68%	0.06%	22305.98	23033.00
tiny255	15	23254.00	949.97	4.09%	-2.40%	22738.00	23857.00
tiny192	15	22969.33	424.10	1.85%	-1.14%	22705.00	23216.00
tiny128	15	23633.33	1556.44	6.59%	-4.07%	22972.98	24748.00
tiny64	15	23448.67	2145.14	9.15%	-3.26%	22624.00	25031.05
tiny32	15	30935.33	4749.53	15.35%	-36.22%	27961.90	33685.05
jdk-15	15	35065.33	5926.67	16.90%	-54.41%	31603.92	38645.07

(a) Unconstrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	28163.57	950.20	3.37%	0.00%	27586.66	28776.68
tiny256K	15	26999.23	1371.41	5.08%	4.13%	26093.73	27878.88
tiny256	15	27677.33	759.05	2.74%	1.73%	27229.00	28139.00
tiny255	15	28073.33	1561.43	5.56%	0.32%	27181.98	29063.08
tiny192	15	28389.33	1223.73	4.31%	-0.80%	27721.00	29163.00
tiny128	15	29337.33	2157.64	7.35%	-4.17%	28113.97	30697.02
tiny64	15	28284.67	2339.65	8.27%	-0.43%	27398.00	29901.12
tiny32	15	41506.00	7977.79	19.22%	-47.37%	37282.00	46737.20
jdk-15	15	35996.67	5026.47	13.96%	-27.81%	33278.97	39245.22

(b) Memory-constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	45335.00	1746.24	3.85%	0.00%	44303.97	46344.03
tiny256K	15	45486.00	1258.88	2.77%	-0.33%	44725.97	46197.00
tiny256	15	46427.00	1121.21	2.41%	-2.41%	45718.98	47017.02
tiny255	15	43617.00	4480.04	10.27%	3.79%	40654.00	45896.05
tiny192	15	44975.00	2720.25	6.05%	0.79%	43171.00	46188.00
tiny128	15	41572.00	5696.64	13.70%	8.30%	38193.97	44874.05
tiny64	15	45525.00	1194.00	2.62%	-0.42%	44801.00	46198.03
tiny32	15	36468.00	5952.91	16.32%	19.56%	33357.00	40315.12
jdk-15	15	37598.00	2558.70	6.81%	17.07%	36104.00	39147.03

(c) CPU-constrained

Table 5.6: Benchmark results of connected_components

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	104943.33	1074.75	1.02%	0.00%	104389.00	105661.02
tiny256K	15	105930.00	1557.88	1.47%	-0.94%	105008.00	106854.00
tiny256	15	105581.33	877.52	0.83%	-0.61%	105085.00	106126.03
tiny255	15	106684.00	7709.32	7.23%	-1.66%	102773.98	111689.08
tiny192	15	106894.67	9454.83	8.84%	-1.86%	102080.98	113613.00
tiny128	15	108378.00	8296.77	7.66%	-3.27%	103933.00	113869.10
tiny64	15	110208.67	7854.08	7.13%	-5.02%	105764.00	115265.03
tiny32	15	135272.67	9591.08	7.09%	-28.90%	129689.98	141025.00
jdk-15	15	185240.00	9500.25	5.13%	-76.51%	179288.85	190610.10

(a) Unconstrained

Table 5.7: Benchmark results of maximal_clique

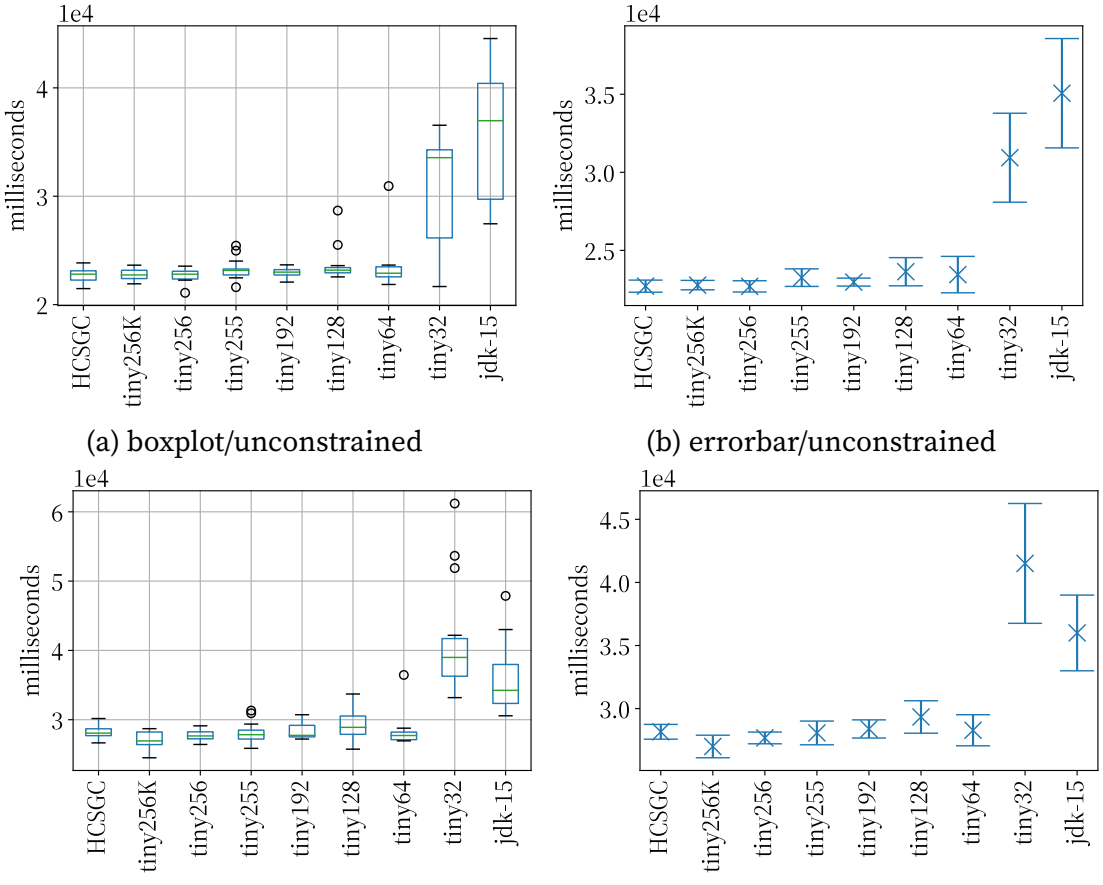
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	138656.00	930.11	0.67%	0.00%	138122.00	139257.00
tiny256K	15	135338.67	1522.61	1.13%	2.39%	134461.00	136264.02
tiny256	15	133948.00	1284.04	0.96%	3.40%	133219.95	134742.05
tiny255	15	132333.33	1194.08	0.90%	4.56%	131538.98	132936.02
tiny192	15	132778.00	1371.23	1.03%	4.24%	132034.98	133663.02
tiny128	15	136513.33	1579.94	1.16%	1.55%	135605.93	137484.02
tiny64	15	134714.00	1892.65	1.40%	2.84%	133692.98	135940.00
tiny32	15	153574.67	6072.85	3.95%	-10.76%	150052.93	157367.10
jdk-15	15	202871.33	11064.45	5.45%	-46.31%	196042.98	209076.22

(b) Memory-constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	177307.00	2119.11	1.20%	0.00%	175900.98	178222.02
tiny256K	15	178375.00	3246.93	1.82%	-0.60%	176218.00	180060.00
tiny256	15	181757.00	1468.83	0.81%	-2.51%	180910.00	182633.00
tiny255	15	171872.22	3637.53	2.12%	3.07%	169968.89	174392.05
tiny192	15	174524.00	7995.86	4.58%	1.57%	169993.00	179577.12
tiny128	15	171645.00	7342.51	4.28%	3.19%	167981.77	176645.02
tiny64	15	174002.00	7954.01	4.57%	1.86%	170015.98	178995.02
tiny32	15	195637.00	5196.44	2.66%	-10.34%	192624.00	198689.00
jdk-15	15	231074.00	8275.23	3.58%	-30.32%	226037.85	235715.10

(c) CPU-constrained

Table 5.7: Benchmark results of maximal_clique



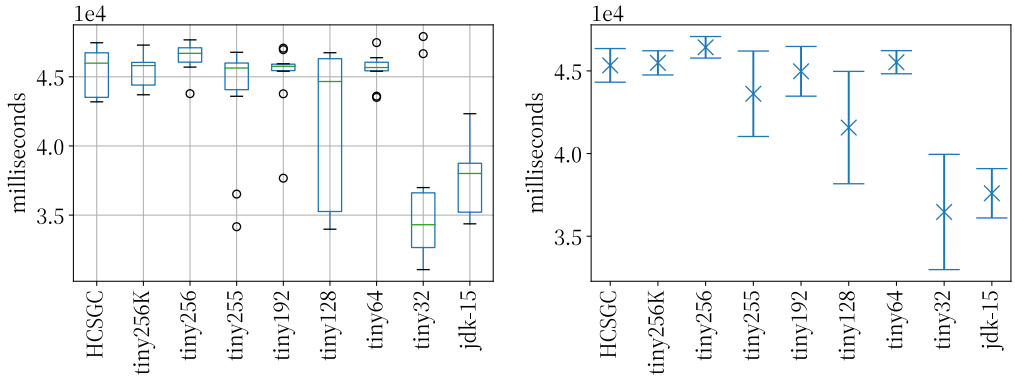
(a) boxplot/unconstrained

(b) errorbar/unconstrained

(c) boxplot/memory-constrained

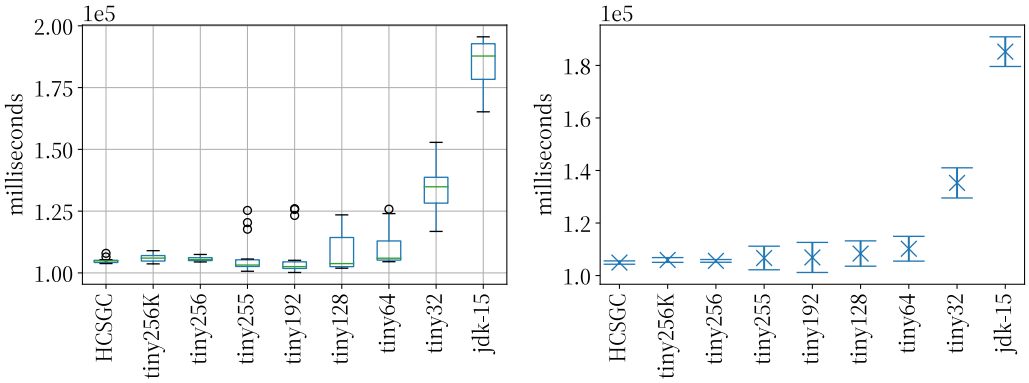
(d) errorbar/memory-constrained

Figure 5.5: Benchmark results of connected_components

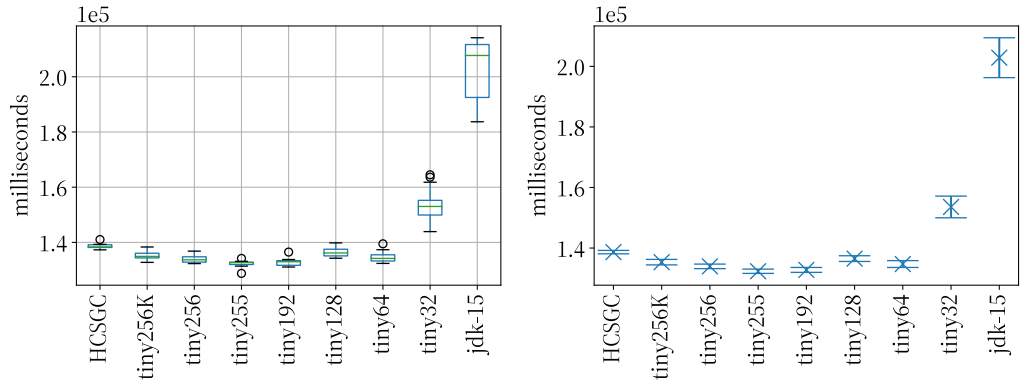


(e) boxplot/CPU-constrained (f) errorbar/CPU-constrained

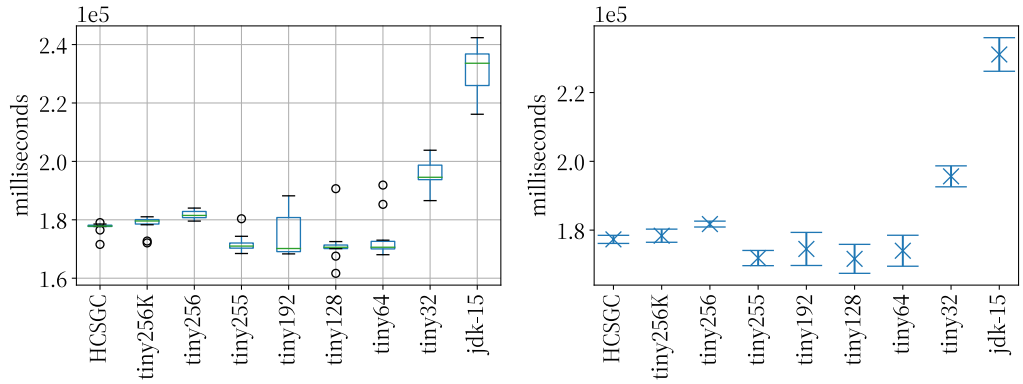
Figure 5.5: Benchmark results of connected_components



(a) boxplot/unconstrained (b) errorbar/unconstrained



(c) boxplot/memory-constrained (d) errorbar/memory-constrained



(e) boxplot/CPU-constrained (f) errorbar/CPU-constrained

Figure 5.6: Benchmark results of maximal_clique

5.3.2 DaCapo Suite

We selected the following benchmarks in the DaCapo suite to test our proposal: `avrora_large`, `fop_default`, `h2_huge`, `luindex_default`, `lusearch_large`, `sunflow_large`, and `xalan_large`. The H2 benchmark with huge input size requires a larger heap. Therefore, the heap size of H2 is set to 6GB, while others are set to 1GB.

As discussed in Section 5.1, although the H2 benchmark requires a large heap, we could not reduce the heap size to constrain the memory. Thus, we will only test the unconstrained and CPU-constrained environment for H2. It also applies to small-heap benchmarks, `avrora_large`, `fop_default`, and `luindex_default`, since they only require a small memory that does not even trigger a single garbage collection. Therefore, reduce the heap size for these benchmarks could not create a memory-constrained environment. The HCSGC does not get better performance on these benchmarks either, since the locality cannot be improved without garbage collection. We only use them to find possible throughput degradation after introducing tiny pages.

Other benchmarks, the `lusearch_large`, `sunflow_large`, and `xalan_large`, will also test in the memory-constrained environment. The heap size is set to 512MB.

The results of the H2 benchmark are shown in Table 1 and visualized in Figure 1. The cache miss rate and results of other benchmarks are listed and plotted in Section A.2. Similar to the JGraphT benchmark, the H2 starts to get performance degradation from `Tiny255` configuration, so the best tiny object threshold for H2 is 256 Bytes. The `sunflow_large` benchmark gets a ~5% regression for all thresholds except 256KB. All other benchmarks have overlapping confidential intervals for all configurations, so we could not find a suitable threshold for these benchmarks. Since these benchmarks do not benefit from both HCSGC and tiny pages, a better option is to turn off both optimizations. We leave it as future work to monitor the GC statistics during runtime and dynamically enable or disable these optimizations.

In CPU-constrained environment, we still get overlapping results for most of the benchmark. The only improvement we get is in the `sunflow_large` benchmark. We get a 0.9% improvement with `Tiny255`, `Tiny192`, and `Tiny128` configuration. In memory-

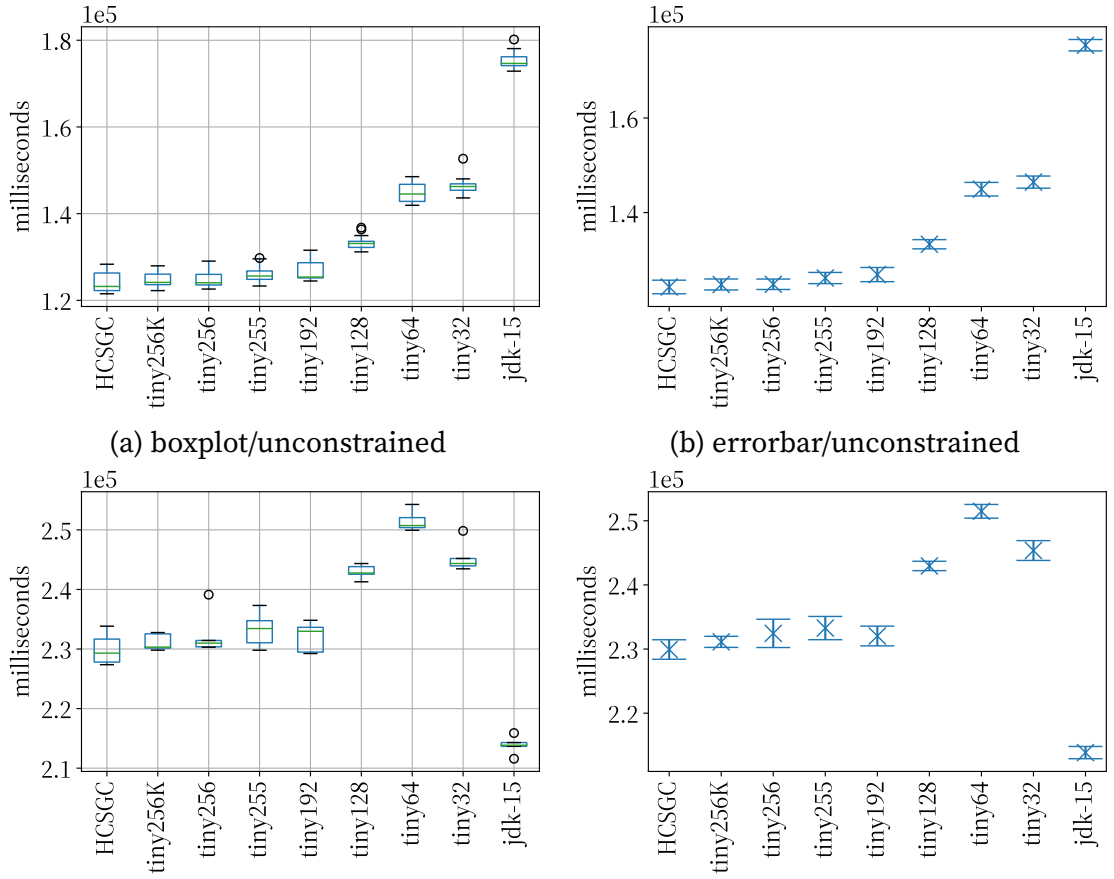
Table 5.8: Benchmark results of H2_huge

(a) Unconstrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	124261.27	2404.87	1.94%	0.00%	122879.29	125776.33
tiny256K	15	124788.96	1936.29	1.55%	-0.42%	123698.62	125989.09
tiny256	15	124831.19	1828.58	1.46%	-0.46%	123827.20	126035.15
tiny255	15	126155.73	1995.13	1.58%	-1.52%	125062.85	127435.28
tiny192	15	126884.23	2529.78	1.99%	-2.11%	125495.10	128497.98
tiny128	15	133299.08	1634.89	1.23%	-7.27%	132381.71	134328.94
tiny64	15	144944.63	2390.78	1.65%	-16.65%	143546.94	146393.26
tiny32	15	146443.12	2155.44	1.47%	-17.85%	145287.16	147847.20
jdk-15	15	175393.85	2007.35	1.14%	-41.15%	174271.18	176684.50

(b) CPU-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	229930.90	2659.21	1.16%	0.00%	228470.77	231444.57
tiny256K	15	231121.32	1414.91	0.61%	-0.52%	230312.20	232018.10
tiny256	15	232444.40	3761.91	1.62%	-1.09%	230604.35	235026.67
tiny255	15	233270.92	2983.87	1.28%	-1.45%	231497.48	235108.95
tiny192	15	232035.04	2525.84	1.09%	-0.92%	230415.61	233539.03
tiny128	15	242957.28	1193.78	0.49%	-5.67%	242214.76	243660.20
tiny64	15	251475.88	1751.05	0.70%	-9.37%	250496.09	252622.57
tiny32	15	245361.32	2580.07	1.05%	-6.71%	243997.32	247100.40
jdk-15	15	213858.16	1545.76	0.72%	6.99%	212889.51	214768.00



(a) boxplot/unconstrained

(b) errorbar/unconstrained

(c) boxplot/CPU-constrained

(d) errorbar/CPU-constrained

Figure 5.7: Benchmark results of H2_huge

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	22130.90	829.41	3.75%	0.00%	21589.84	22557.73
Tiny256	10	21644.80	2268.92	10.48%	-2.20%	20129.94	22779.10
Tiny192	10	22394.00	1064.42	4.75%	1.19%	21661.29	22857.10
Tiny128	10	22537.30	1059.61	4.70%	1.84%	21839.09	23009.00
ZGC	10	23009.00	910.39	3.96%	3.97%	22415.85	23447.90

(a) critical-jOPS

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	26126.60	854.41	3.27%	0.00%	25583.40	26556.11
Tiny256	10	26296.70	972.62	3.70%	0.65%	25639.30	26727.00
Tiny192	10	26312.20	1061.73	4.04%	0.71%	25597.60	26783.60
Tiny128	10	26383.50	1336.10	5.06%	0.98%	25496.10	26984.40
ZGC	10	26183.80	1019.07	3.89%	0.22%	25526.40	26641.20

(b) max-jOPS

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	10	26571.90	708.97	2.67%	0.00%	26136.30	26963.50
Tiny256	10	27221.60	627.31	2.30%	2.45%	26865.00	27607.01
Tiny192	10	27538.20	823.20	2.99%	3.64%	27011.58	27953.03
Tiny128	10	27163.90	1060.61	3.90%	2.23%	26504.44	27668.40
ZGC	10	26704.40	1033.21	3.87%	0.50%	26024.24	27218.20

(c) HBIR

Table 5.9: Benchmark results of SPECjbb2015

constrained environment, we get similar results to the unconstrained one.

5.3.3 SPECjbb2015

A single iteration of SPECjbb2015 will last longer than two hours. Only a subset of the configurations are selected due to time limitations: the HCSGC, the vanilla ZGC, and tiny page implementation with 256 Bytes, 192 Bytes, 128 Bytes as the tiny object threshold. All configurations will run with the composite setting of SPECjbb2015 with 64G heap size and without any constraints since the benchmark could exhaust the hardware.

The SPECjbb reports two metrics, `max-jOPS` and `critical-jOPS`, corresponding to throughput and latency. We also recorded the HBIR for each iteration. The results are shown in Table 5.9 and Figure A.10, higher is better in all cases. We get a over 3% larger HBIR comparing to both ZGC and HCSGC. But the confidence intervals of the two metrics are overlapping, we cannot say whether the tiny page improves the

overall performance.

5.3.4 Analysis

First, we will analyze benchmarks that do not get improvements, or even have worse performance with our tiny page implementation.

In most benchmarks that the HCSGC does not get improvements, our implementation also failed to improve. In this case, all configurations have overlapping confidence intervals. These benchmarks could be grouped to two categories: (1) small-heap benchmarks, the `avrora_large`, `fop_default`, and `luindex_default`, (2) large-heap benchmarks, the `lusearch_large`, `xalan_large`, and `SPECjbb2015`. As described in Section 5.3.2, small-heap benchmarks will not trigger garbage collection. HCSGC needs garbage collections to improve the locality, and our tiny page implementation aims to optimize the relocation phase of garbage collection, so if no GC happened, no improvements could be found in both implementations.

If the benchmark requires a large heap but still does not benefit from HCSGC, a possible reason is that most of the objects in that benchmark die young. HCSGC could only improve the locality if the object lives long enough, so the locality optimization will not work for these benchmarks. In benchmarks such as `lusearch_large`, the locality is not improved, but HCSGC does not get worse performance although it introduces more relocation and fragmentation. It means the introduced overhead is not large enough to impact the overall performance. Our tiny page implementation could reduce the relocation amount and fragmentation, but it is meaningless when these overheads are indistinguishable. Furthermore, if most objects die young but heap size is large, the allocation amount is also likely to be large. Since the tiny page proposal adds an extra overhead in the object allocation phase, we may even see small regressions in these benchmarks.

In unconstrained environments, our implementation does not get better performance than HCSGC. This is reasonable since we aim to reduce the relocation amount and memory fragmentation, which will not affect the performance in unconstrained environments. Running these tests could help us find the threshold that objects starting to get benefits from locality. For all the benchmarks we tested, 256 Bytes is the best threshold for the tiny objects.

Second, we will analyze improvements in the `sunflow_large` and `maximal_clique` benchmark.

Another possible reason for large heap benchmarks which do not benefit from HCSGC is that the fragmentation introduced by HCSGC outperforms the locality benefits and lowers the throughput. In the CPU-constrained test of `sunflow_large` and `connected_components` benchmarks, the HCSGC has the same or worse performance. If we reduce the relocation amount, we could see an improvement. The regression of `sunflow_large` in unconstrained environments could also be explained, as we reduced a bit locality to trade for a better relocation performance. It will cause a lower throughput in unconstrained environments but an improvement in constrained ones.

The `maximal_clique` is the ideal benchmark for our implementation because the HCSGC gets a huge improvement over ZGC, and the tiny page could further increase the throughput in constrained environments. We will analyze it by recording detailed information about each garbage collection.

5.4 The relocation performance

Lastly, we want to find some relations between the threshold of tiny pages, the GC status, and the throughput. Since we get performance improvement in both CPU- and memory-constrained environment of `maximal_clique` benchmark, we choose it again for the analysis.

Three different environment configurations are used to show the GC status under different constraints, named *JGraphT 1G*, *JGraphT 512M*, and *JGraphT 1G (taskset)*. 1GB is sufficient for the maximal clique algorithm to run. Thus, the heap size is set to 1GB to create an unconstrained environment in the first configuration. It is set to 512M in the second one to emulate a memory-constrained environment. In the last configuration, enough heap size is provided, but CPU is limited using the `taskset` command mentioned in Section 4.1.3.

The result is shown in Table 5.10. The table contains the following data (from left to right): the tiny object threshold in Bytes, the average of wall-clock execution time, count of GC cycles, the proportion of tiny pages to all pages, and relocation status. In the relocation status part, *Lazy* means the Lazy relocation, *Full* means the Full relocation, and

Total is the summarize of them. *Mutator* indicates that the relocation happened in the worker thread and will compete with mutators, and *GC* indicates that it happened in the concurrent GC thread. *Count* is measured by relocation operations per second, while *Size* is relocation amounts in MBytes per second.

From the results, we could summarize the following points.

- For all three configurations, the total relocation handled by mutator threads reduces with a lower threshold, which leads to a higher relocation performance.
- In memory-constrained environments, the memory fragmentation introduced by HCSGC will drastically increase the memory usage and count of GC cycles. Although ZGC is a mostly concurrent collector, more GC will increase the system load and slow down the program. More relocation on the critical path will also result in the same thing. Therefore, we see improvements in a lower threshold.
- In CPU-constrained environments, the memory fragmentation will not increase the GC count. However, in this case, relocations handled by the concurrent GC thread will compete with the mutator thread. Thus, we still see improvements when fewer objects are selected for relocation.
- In unconstrained environments, the memory fragmentation will not increase the GC count, and the relocations in GC thread will not infect the performance either. As a result, a lower threshold will only cause a poorer locality and a lower throughput.

Tiny object limit (Bytes)	Time(s)	GC Count	Tiny page %	Lazy		Full				Total			
				Mutator		Mutator		GC		Mutator		GC	
				Count	Size	Count	Size	Count	Size	Count	Size	Count	Size
JGraphT 1G													
262144	59.66	40.2	94.0	83,547	2	170,277	4.2	155,178	5.4	253,824	6.2	155,178	5.4
256	58.47	41.0	89.6	88,255	2	167,267	4	168,973	6.8	255,522	6	168,973	6.8
255	67.09	37.8	10.4	185,208	5	22,527	0	122,685	5	207,735	5	122,694	5
192	65.88	37.4	10.4	185,757	4.6	22,970	0	127,587	5.2	208,727	4.6	127,587	5.2
128	66.74	37.4	10.6	180,762	4.2	24,303	0.2	127,042	5	205,065	4.4	127,042	5
64	81.95	36.6	8.4	117,551	2.4	20,424	0.2	95,981	3.6	137,974	2.6	95,981	3.6
32	83.40	35.2	4.8	123,466	2.6	23,181	0.2	103,024	3	146,647	2.8	103,024	3
0	95.30	34.4	0.0	-	0	10,087	0	49,661	1.8	10,087	0	49,661	1.8
JGraphT 512M													
262144	106.83	167.2	89.8	114,123	2.6	311,975	7.8	1,191,542	48	426,097	10.4	1,191,542	48
256	107.76	167.0	81.2	112,344	2.6	312,640	7.8	1,558,428	56	424,984	10.4	1,558,428	56
255	87.31	115.3	21.5	369,727	9.3	19,786	0	125,206	5.8	389,513	9.3	125,206	5.8
192	92.21	111.2	20.8	328,967	8.2	20,236	0	106,601	4.8	349,203	8.2	106,601	4.8
128	93.78	113.0	20.8	322,712	8	17,434	0	111,050	5	340,147	8	111,050	5
64	95.84	106.4	19.4	311,319	7.4	17,350	0	109,256	5.2	328,669	7.4	109,256	5.2
32	108.76	93.0	6.4	205,973	4.2	17,119	0	110,301	3.6	223,092	4.2	110,301	3.6
0	102.26	87.8	0.0	-	0	21,276	0.2	58,434	2.2	21,276	0.2	58,434	2.2
JGraphT 1G (taskset)													
262144	88.59	61.2	94.0	52,086	1	186,966	4.4	98,410	3.6	239,052	5.4	98,410	3.6
256	88.95	60.8	89.2	52,548	1	180,143	4.6	105,329	4	232,692	5.6	105,329	4
255	86.86	54.6	11.8	199,147	4.6	17,143	0	75,589	3	216,290	4.6	75,589	3
192	83.93	54.4	12.2	207,769	5	16,513	0	79,984	3.2	224,282	5	79,984	3.2
128	95.24	53.2	11.4	164,539	3.8	15,441	0	67,816	2.4	179,980	3.8	67,816	2.4
64	93.98	53.6	11.4	168,900	3.8	18,180	0	71,683	2.8	187,080	3.8	71,683	2.8
32	98.21	54.0	3.8	152,594	3	26,101	0.4	74,049	2.2	178,695	3.4	74,049	2.2
0	88.20	48.4	0.0	-	0	31,502	0.2	48,233	1.8	31,502	0.2	48,233	1.8

Table 5.10: GC status for JGraphT

Chapter 6

Conclusions and Future work

Introducing more relocations to objects based on the access pattern could increase the locality and overall throughput, but with the degradation of relocation performance. This thesis shows a feasible way to improve the relocation performance by ignoring larger objects in the relocation phase. We achieved to reduce more than 40% of the relocations without affecting the overall throughput. Also, we see a 3-5% overall performance improvement when running the benchmark in constrained environments. The goal of improving the relocation performance is thus considered to be fulfilled.

From a broader perspective, this project is meaningful for both application and research purposes. The relocation of reduced HCSGC can increase the performance in constrained environments, which means our implementation is able to reduce the power consumption and also speed up the program in such environments, for example, embedded devices. Embedded systems providers might want to utilize this GC for the sake of performance and cost. Also, we run multiple benchmarks to find the balance between the cache locality and relocation amount. Thus, this project can be a good reference for those who develop next-generation GCs.

Our proposal of tiny pages could not get improvements in most cases when HCSGC does not. The HCSGC depends on applications exhibiting stable access patterns on long-lived objects. If objects die young, the HCSGC could not get many locality benefits. At the same time, more short-lived objects also mean more allocations. The extra check for tiny pages at the allocation stage may lower the performance.

As future work, a backing-off scheme can be implemented to avoid regression in some cases. An extra check of object survival rate can be added to the relocation phase. If the survival rate is lower enough, turn off the HCSGC as well as the tiny page. This will cause all tiny and small objects to be placed on tiny pages. As proposed before, tiny and small pages are identical from vanilla JDK's perspective, so it is reasonable to do this when HCSGC is turned off.

We also implemented both HCSGC and tiny pages in the newer version of OpenJDK, namely JDK17. However, the DaCapo benchmark got around 10% regression in the new version, and some of the benchmarks even failed to run at all (such as tradesoap and tradebeans). To keep coherent with previous HCSGC implementation, we decided to continue using JDK15.

The DaCapo benchmark using reflections to invoke corresponding programs to run the test, but OpenJDK became more strict in unsafe operations like this. The latest version of DaCapo was released more than 10 years ago and it is reasonable that it could not run on modern platforms. So another future work is to develop a benchmark that could run in modern versions of JDK, and have a coherent performance with older versions.

References

[1] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*. Cambridge: Cambridge University Press, 2002. ISBN 9780521820608

[2] "HotSpot Virtual Machine Garbage Collection Tuning Guide." [Online]. Available: https://docs.oracle.com/en/java/javase/15/gctuning/index.html

[3] "The Parallel Collector." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[4] "Concurrent Mark Sweep (CMS) Collector." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[5] P. Lidén and S. Karlsson, "JEP 333: ZGC: A Scalable Low-Latency Garbage Collector." [Online]. Available: https://openjdk.java.net/jeps/333

[6] C. H. Flood and R. Kennke, "JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector." [Online]. Available: https://openjdk.java.net/jeps/189

[7] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The Garbage Collection Advantage: Improving Program Locality," *SIGPLAN Not.*, vol. 39, no. 10, p. 69–80, Oct. 2004. doi: 10.1145/1035292.1028983. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/1035292.1028983

[8] A. M. Yang, E. Österlund, and T. Wrigstad, "Improving Program Locality in the GC Using Hotness," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and*

Implementation, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3385412.3385977. ISBN 9781450376136 p. 301–313. [Online]. Available: https://doi.org/10.1145/3385412.3385977

[9] W.-k. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang, "Profile-Guided Proactive Garbage Collection for Locality Optimization," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: Association for Computing Machinery, 2006. doi: 10.1145/1133981.1134021. ISBN 1595933204 p. 332–340. [Online]. Available: https://doi.org/10.1145/1133981.1134021

[10] "OpenJDK License: GPLv2 with the Classpath Exception." [Online]. Available: https://openjdk.java.net/legal/gplv2+ce.html

[11] "OpenJDK - How to contribute." [Online]. Available: https://openjdk.java.net/contribute/

[12] H. Schildt, *Java*, 6th ed. McGraw Hill Osborne, 2014. ISBN 0-07-180925-2

[13] S. Oaks, *Java performance : the definitive guide*, 1st ed. Sebastopol, California : O'Reilly Media, 2014. ISBN 1-4493-6354-7

[14] "HotSpot Java Virtual Machine Guide." [Online]. Available: https://docs.oracle.com/en/java/javase/15/vm/index.html

[15] "JEP 318: Epsilon: A No-Op Garbage Collector (Experimental)." [Online]. Available: https://openjdk.java.net/jeps/318

[16] "The java Command." [Online]. Available: https://docs.oracle.com/en/java/javase/15/docs/specs/man/java.html

[17] "openjdk/jdk: JDK main-line development." [Online]. Available: https://github.com/openjdk/jdk

[18] R. Courts, "Improving locality of reference in a garbage-collecting memory management system," *Commun. ACM*, vol. 31, no. 9, p. 1128–1138, sep 1988. doi: 10.1145/48529.48536. [Online]. Available: https://doi.org/10.1145/48529.48536

[19] M. S. Lam, P. R. Wilson, and T. G. Moher, "Object type directed garbage collection to improve locality," in *Memory Management*, Y. Bekkers and J. Cohen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992. ISBN 978-3-540-47315-2 pp. 404–425.

[20] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07. New York, NY, USA: Association for Computing Machinery, 2007. doi: 10.1145/1248377.1248394. ISBN 9781595936677 p. 93–104. [Online]. Available: https://doi.org/10.1145/1248377.1248394

[21] M. Heinrichs, "Java and the modern CPU, Part 1: Memory and the cache hierarchy." [Online]. Available: https://blogs.oracle.com/javamagazine/java-and-the-modern-cpu-part-1-memory-and-the-cache-hierarchy

[22] " Software Optimization Guide for AMD Family 17h Processors." [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf

[23] " Intel® 64 and IA-32 Architectures Optimization Reference Manual." [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[24] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006. doi: 10.1145/1167473.1167488. ISBN 1595933484 p. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[25] "JGraphT: a Java library of graph theory data structures and algorithms." [Online]. Available: https://jgrapht.org/

[26] R. Samudrala and J. Moult, "A graph-theoretic algorithm for comparative modeling of protein structure11edited by f. cohen," *Journal of Molecular Biology*, vol. 279, no. 1, pp. 287–302, 1998. doi: https://doi.org/10.1006/jmbi.1998.1689. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022283698916898

[27] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, p. 372–378, Jun. 1973. doi: 10.1145/362248.362272. [Online]. Available: https://doi.org/10.1145/362248.362272

[28] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[29] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[30] A. M. Yang and T. Wrigstad, "TobiasWrigstad/pldi2020-artefact." [Online]. Available: https://github.com/TobiasWrigstad/pldi2020-artefact

[31] " Standard Performance Evaluation Corporation: SPECjbb® 2015." [Online]. Available: https://www.spec.org/jbb2015/

[32] "SPECjbb2015 Benchmark Design Document." [Online]. Available: https://www.spec.org/jbb2015/docs/designdocument.pdf

Appendix A

Extra results

A.1 JGraphT

Cache miss rate data of JGraphT benchmark is presented in this part. The data is collected with `perf stat -e L1-dcache-loads, L1-dcache-load-misses,LLC-loads,LLC-load-misses`, and cache miss rate is calculated with Equation A.1.

$$cache\_miss\_rate = \frac{cache\_misses}{cache\_loads} \times 100\% \tag{A.1}$$

The cache miss rate of connected_components is presented in Table A.1 and visualized in Figure A.1.

The cache miss rate of maximal_clique is presented in Table A.2 and visualized in Figure A.2.

A.2 DaCapo

The cache miss rate of `H2_huge` benchmark is presented in Table A.3 and Figure A.3. Results of other benchmarks in DaCapo suite are presented in Tables A.4 to A.9 and Figures A.4 to A.9.

A.3 SPECjbb2015

The benchmark results of `SPECjbb2015` are plotted in Figure A.10.

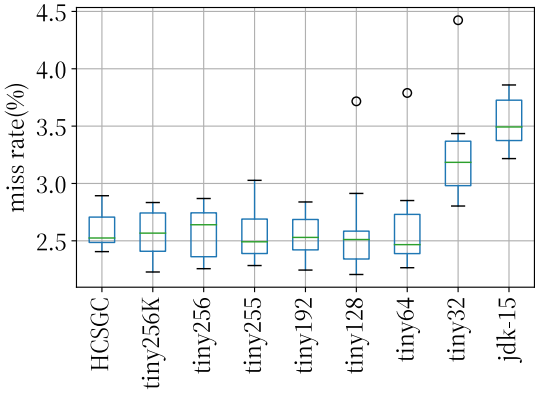
Table A.1: Cache miss rate in connected_components

(a) L1 cache miss rate

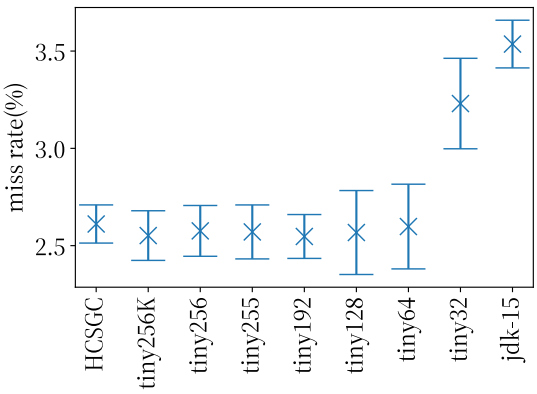
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Change	CI Lower	CI Upper
HCSGC	15	2.61	0.17	6.32%	0.00%	2.52	2.71
tiny256K	15	2.55	0.21	8.32%	-2.28%	2.42	2.68
tiny256	15	2.58	0.22	8.53%	-1.36%	2.44	2.71
tiny255	15	2.57	0.23	8.98%	-1.56%	2.44	2.72
tiny192	15	2.55	0.19	7.39%	-2.46%	2.43	2.66
tiny128	15	2.57	0.37	14.27%	-1.68%	2.39	2.82
tiny64	15	2.60	0.38	14.59%	-0.50%	2.42	2.87
tiny32	15	3.23	0.39	12.01%	23.70%	3.04	3.50
jdk-15	15	3.54	0.20	5.78%	35.41%	3.41	3.66

(b) LLC cache miss rate

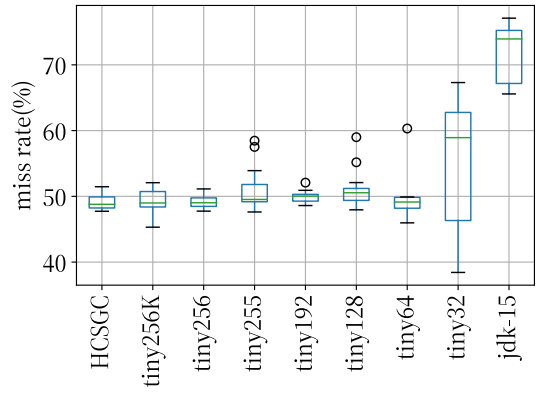
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Change	CI Lower	CI Upper
HCSGC	15	49.21	1.24	2.52%	0.00%	48.52	49.95
tiny256K	15	49.18	1.91	3.87%	-0.06%	48.03	50.30
tiny256	15	49.09	0.93	1.89%	-0.23%	48.55	49.68
tiny255	15	51.00	3.27	6.41%	3.64%	49.28	53.13
tiny192	15	49.91	0.92	1.84%	1.43%	49.39	50.46
tiny128	15	50.93	2.85	5.60%	3.51%	49.50	52.82
tiny64	15	49.46	3.27	6.60%	0.51%	47.98	51.71
tiny32	15	54.95	9.78	17.80%	11.66%	49.01	60.64
jdk-15	15	71.50	4.30	6.01%	45.31%	68.99	74.03



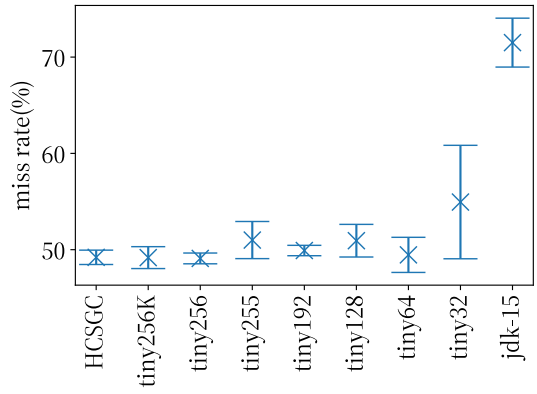
(a) boxplot/L1 miss rate



(b) errorbar/L1 miss rate



(c) boxplot/LLC miss rate



(d) errorbar/LLC miss rate

Figure A.1: Cache miss rate in connected_components

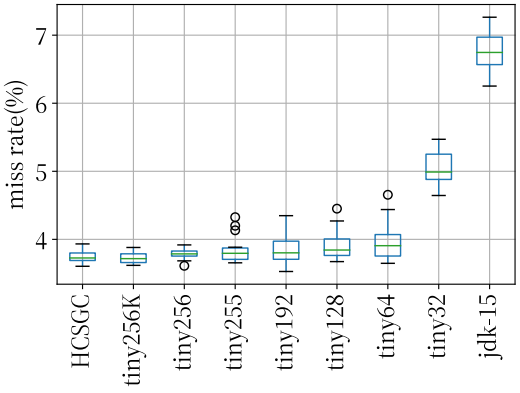
Table A.2: Cache miss rate in maximal_clique

(a) L1 cache miss rate

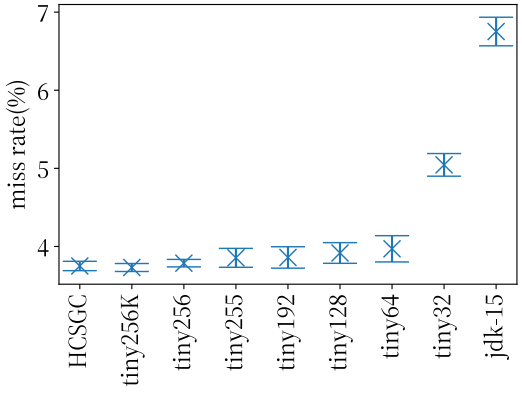
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Change	CI Lower	CI Upper
HCSGC	15	3.75	0.10	2.71%	0.00%	3.69	3.81
tiny256K	15	3.73	0.09	2.29%	-0.52%	3.68	3.78
tiny256	15	3.79	0.08	2.11%	0.97%	3.74	3.83
tiny255	15	3.85	0.20	5.32%	2.79%	3.74	3.99
tiny192	15	3.86	0.23	6.03%	2.91%	3.73	4.01
tiny128	15	3.92	0.22	5.67%	4.44%	3.80	4.06
tiny64	15	3.97	0.28	7.17%	5.87%	3.82	4.16
tiny32	15	5.04	0.25	4.88%	34.53%	4.90	5.19
jdk-15	15	6.75	0.31	4.52%	80.05%	6.57	6.93

(b) LLC cache miss rate

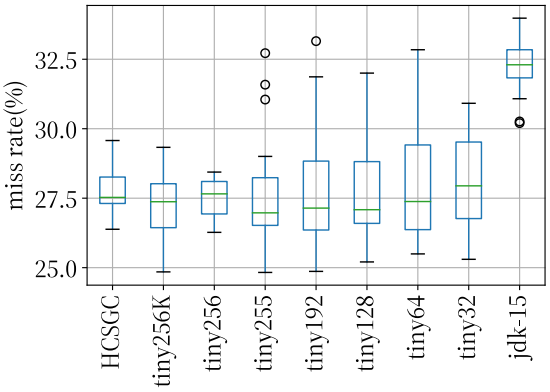
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Change	CI Lower	CI Upper
HCSGC	15	27.75	0.81	2.90%	0.00%	27.28	28.25
tiny256K	15	27.29	1.21	4.43%	-1.66%	26.56	28.00
tiny256	15	27.53	0.70	2.54%	0.80%	27.11	27.94
tiny255	15	27.75	2.30	8.27%	0.00%	26.51	29.21
tiny192	15	27.91	2.46	8.81%	0.58%	26.54	29.49
tiny128	15	27.83	2.03	7.30%	0.27%	26.68	29.11
tiny64	15	28.05	2.44	8.70%	1.08%	26.71	29.58
tiny32	15	28.23	1.67	5.93%	1.71%	27.23	29.24
jdk-15	15	32.19	1.05	3.25%	15.99%	31.53	32.78



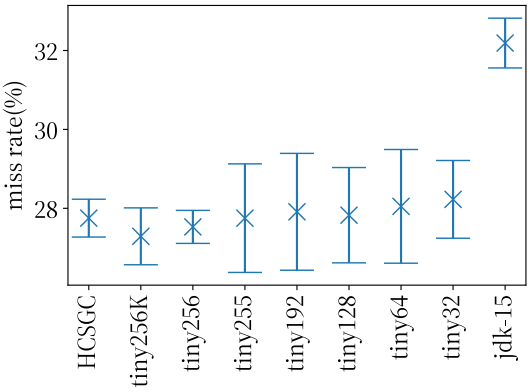
(a) boxplot/L1 miss rate



(b) errorbar/L1 miss rate



(c) boxplot/LLC miss rate



(d) errorbar/LLC miss rate

Figure A.2: Cache miss rate in connected_components

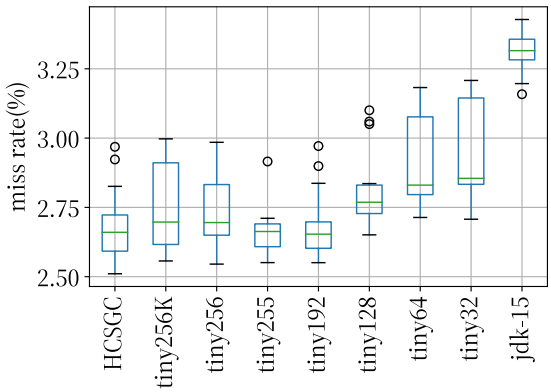
Table A.3: Cache miss rate of H2_huge

(a) L1 cache miss rate

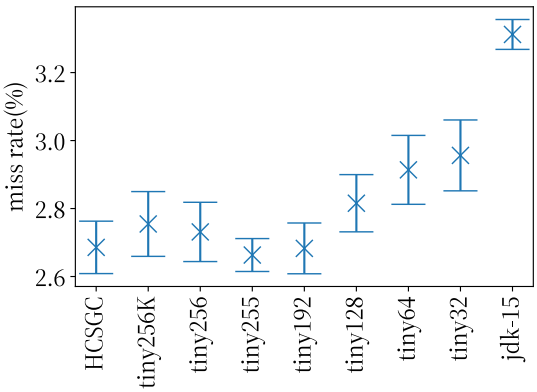
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	2.69	0.13	4.87%	0.00%	2.61	2.77
tiny256K	15	2.75	0.16	5.78%	-2.56%	2.66	2.85
tiny256	15	2.73	0.14	5.30%	-1.69%	2.65	2.82
tiny255	15	2.66	0.08	3.14%	0.84%	2.62	2.72
tiny192	15	2.68	0.13	4.70%	0.11%	2.61	2.76
tiny128	15	2.82	0.14	5.02%	-4.84%	2.74	2.91
tiny64	15	2.91	0.17	5.90%	-8.49%	2.82	3.02
tiny32	15	2.96	0.18	5.97%	-10.08%	2.85	3.06
jdk-15	15	3.31	0.07	2.20%	-23.33%	3.27	3.35

(b) CPU-Constrained

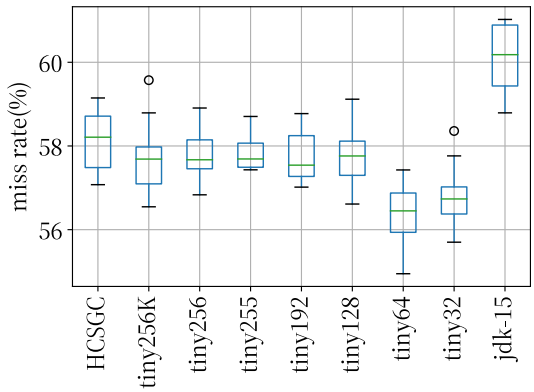
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	58.14	0.72	1.24%	0.00%	57.70	58.56
tiny256K	15	57.68	0.83	1.43%	0.79%	57.21	58.21
tiny256	15	57.82	0.55	0.95%	0.54%	57.51	58.16
tiny255	15	57.86	0.47	0.80%	0.47%	57.61	58.16
tiny192	15	57.74	0.58	1.00%	0.68%	57.40	58.09
tiny128	15	57.71	0.71	1.24%	0.73%	57.28	58.14
tiny64	15	56.38	0.76	1.34%	3.02%	55.91	56.81
tiny32	15	56.79	0.68	1.20%	2.31%	56.40	57.22
jdk-15	15	60.06	0.80	1.33%	-3.31%	59.58	60.53



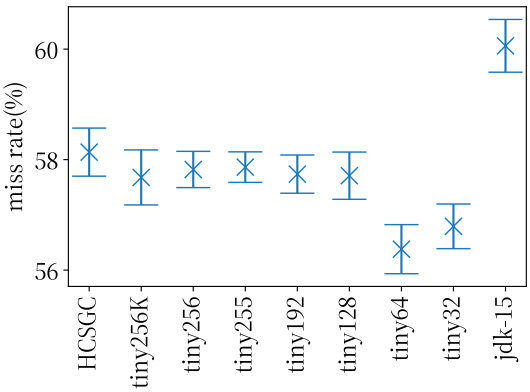
(a) boxplot/L1 miss rate



(b) errorbar/L1 miss rate



(c) boxplot/LLC miss rate



(d) errorbar/LLC miss rate

Figure A.3: Cache miss rate of H2_huge

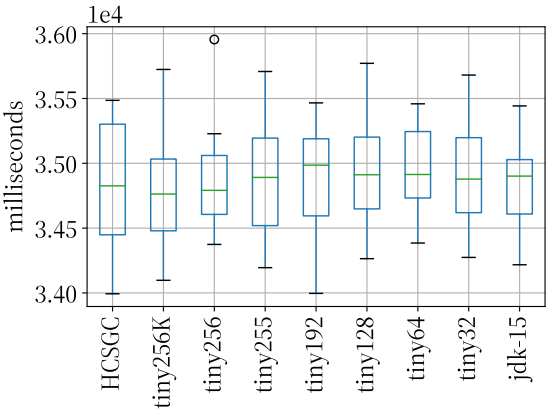
Table A.4: Benchmark results of avrora_large

(a) Unconstrained

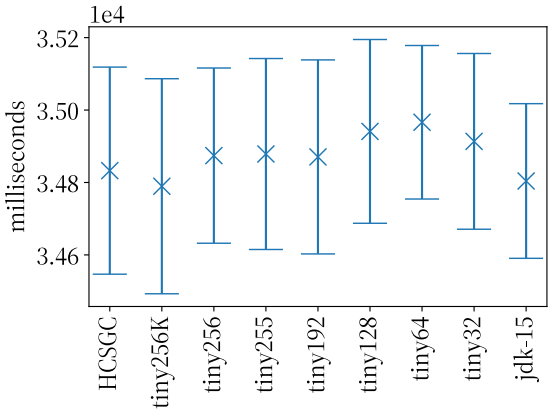
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	34832.71	479.83	1.38%	0.00%	34543.31	35118.96
tiny256K	15	34789.61	502.72	1.45%	0.12%	34504.44	35095.75
tiny256	15	34874.18	402.64	1.15%	-0.12%	34654.81	35143.10
tiny255	15	34878.65	445.90	1.28%	-0.13%	34619.25	35148.53
tiny192	15	34870.61	450.66	1.29%	-0.11%	34590.41	35121.00
tiny128	15	34941.09	421.57	1.21%	-0.31%	34691.04	35202.60
tiny64	15	34966.29	355.43	1.02%	-0.38%	34750.37	35175.12
tiny32	15	34913.54	404.48	1.16%	-0.23%	34674.78	35159.16
jdk-15	15	34804.11	362.06	1.04%	0.08%	34583.52	35016.66

(b) CPU-Constrained

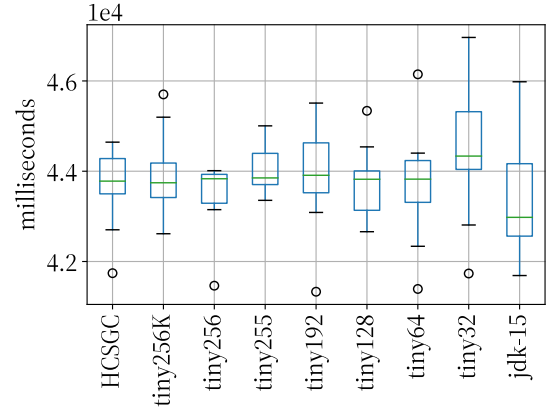
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	43664.13	886.41	2.03%	0.00%	43109.32	44128.83
tiny256K	15	43922.03	934.01	2.13%	-0.59%	43399.58	44493.89
tiny256	15	43499.89	783.94	1.80%	0.38%	42955.81	43869.81
tiny255	15	44042.13	519.26	1.18%	-0.87%	43753.22	44361.76
tiny192	15	43871.45	1155.14	2.63%	-0.47%	43135.38	44508.42
tiny128	15	43748.44	805.24	1.84%	-0.19%	43298.06	44236.90
tiny64	15	43693.20	1265.40	2.90%	-0.07%	42958.33	44445.15
tiny32	15	44429.16	1473.13	3.32%	-1.75%	43558.25	45262.33
jdk-15	15	43480.42	1460.16	3.36%	0.42%	42670.94	44370.31



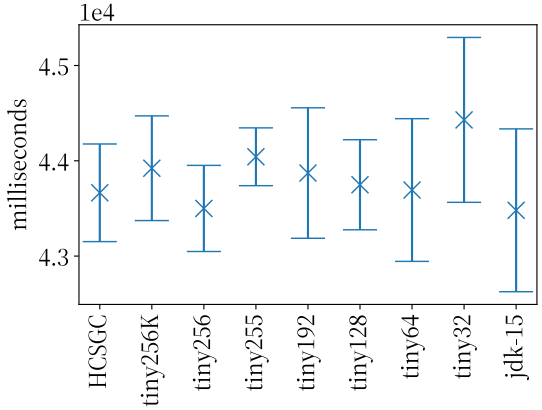
(a) boxplot/Unconstrained



(b) errorbar/Unconstrained



(c) boxplot/CPU-constrained



(d) errorbar/CPU-constrained

Figure A.4: Benchmark results of avrora_large

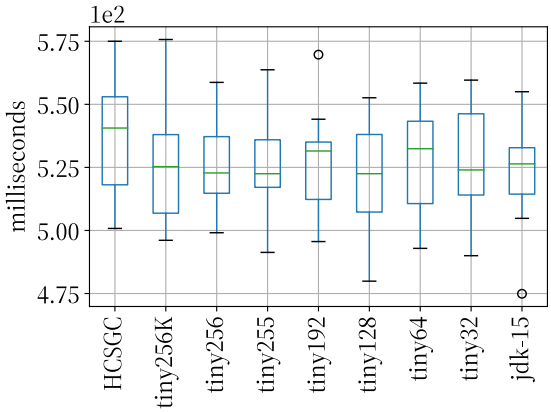
Table A.5: Benchmark results of fop_default

(a) Unconstrained

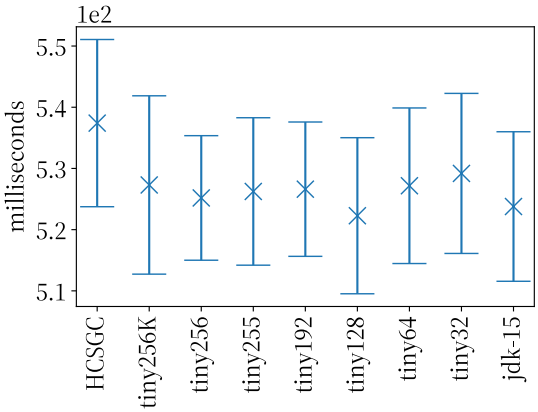
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	537.41	22.61	4.21%	0.00%	524.14	551.15
tiny256K	15	527.30	24.12	4.57%	1.88%	513.17	542.62
tiny256	15	525.18	16.90	3.22%	2.28%	515.29	535.48
tiny255	15	526.24	20.22	3.84%	2.08%	514.33	538.42
tiny192	15	526.61	18.27	3.47%	2.01%	516.33	538.14
tiny128	15	522.27	21.60	4.14%	2.82%	508.82	534.76
tiny64	15	527.17	21.25	4.03%	1.90%	514.16	539.28
tiny32	15	529.19	22.17	4.19%	1.53%	515.85	542.15
jdk-15	15	523.78	20.35	3.89%	2.54%	510.98	535.45

(b) CPU-Constrained

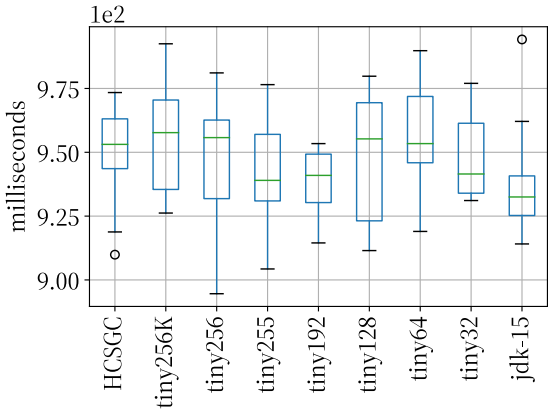
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	949.12	20.88	2.20%	0.00%	936.08	960.50
tiny256K	15	955.99	22.53	2.36%	-0.72%	942.86	969.46
tiny256	15	947.18	25.37	2.68%	0.20%	931.34	960.86
tiny255	15	943.46	21.65	2.29%	0.60%	930.51	955.94
tiny192	15	938.81	13.45	1.43%	1.09%	930.73	946.38
tiny128	15	948.43	25.30	2.67%	0.07%	933.23	963.10
tiny64	15	955.98	21.94	2.29%	-0.72%	943.12	968.78
tiny32	15	947.84	16.41	1.73%	0.13%	938.63	957.78
jdk-15	15	939.06	23.45	2.50%	1.06%	927.31	954.25



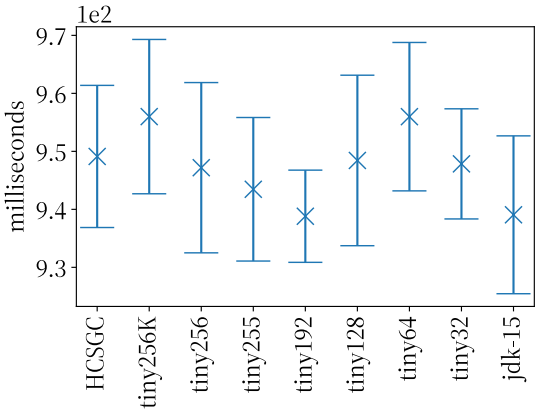
(a) boxplot/Unconstrained



(b) errorbar/Unconstrained



(c) boxplot/CPU-constrained



(d) errorbar/CPU-constrained

Figure A.5: Benchmark results of fop_default

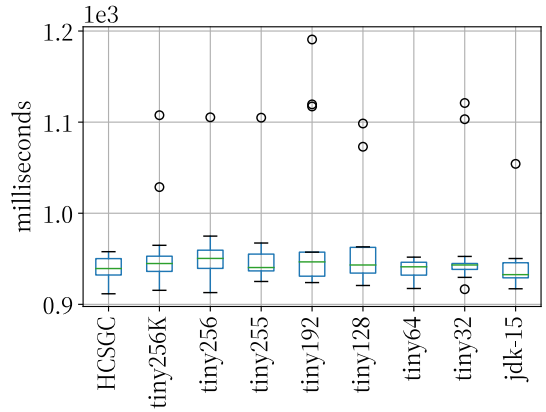
Table A.6: Benchmark results of luindex_default

(a) Unconstrained

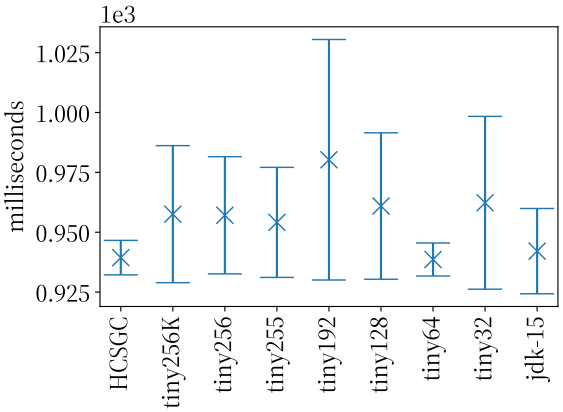
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	939.39	12.02	1.28%	0.00%	931.78	946.10
tiny256K	15	957.54	49.27	5.15%	-1.93%	935.21	992.76
tiny256	15	957.07	44.00	4.60%	-1.88%	938.12	988.11
tiny255	15	954.11	43.54	4.56%	-1.57%	937.22	984.98
tiny192	15	980.27	86.13	8.79%	-4.35%	937.09	1036.00
tiny128	15	960.95	52.55	5.47%	-2.29%	936.45	997.05
tiny64	15	938.61	11.62	1.24%	0.08%	931.28	945.29
tiny32	15	962.29	61.50	6.39%	-2.44%	935.87	1006.92
jdk-15	15	942.11	32.64	3.46%	-0.29%	928.93	965.33

(b) CPU-Constrained

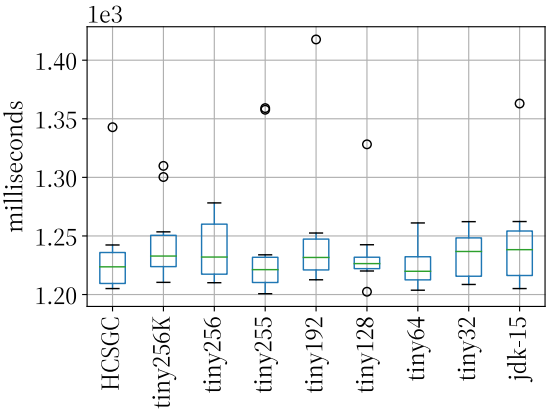
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	1233.48	40.57	3.29%	0.00%	1215.20	1260.82
tiny256K	15	1245.34	33.69	2.71%	-0.96%	1227.36	1266.79
tiny256	15	1238.76	25.35	2.05%	-0.43%	1224.46	1253.96
tiny255	15	1245.23	60.43	4.85%	-0.95%	1214.49	1286.01
tiny192	15	1249.24	60.71	4.86%	-1.28%	1224.34	1289.89
tiny128	15	1235.46	34.18	2.77%	-0.16%	1220.20	1258.14
tiny64	15	1224.41	17.30	1.41%	0.74%	1214.93	1235.28
tiny32	15	1234.34	19.56	1.58%	-0.07%	1222.97	1245.89
jdk-15	15	1245.65	45.75	3.67%	-0.99%	1223.17	1275.79



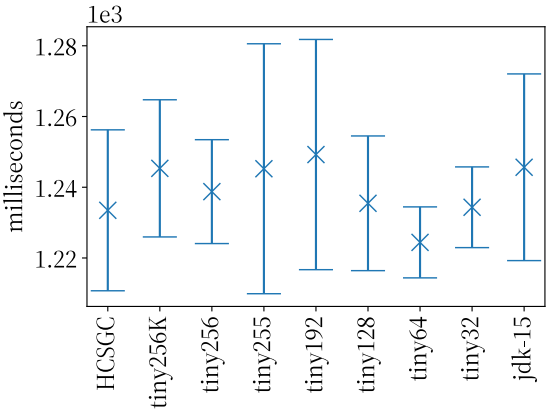
(a) boxplot/Unconstrained



(b) errorbar/Unconstrained



(c) boxplot/CPU-constrained



(d) errorbar/CPU-constrained

Figure A.6: Benchmark results of luindex_default

Table A.7: Benchmark results of lusearch_large

(a) Unconstrained

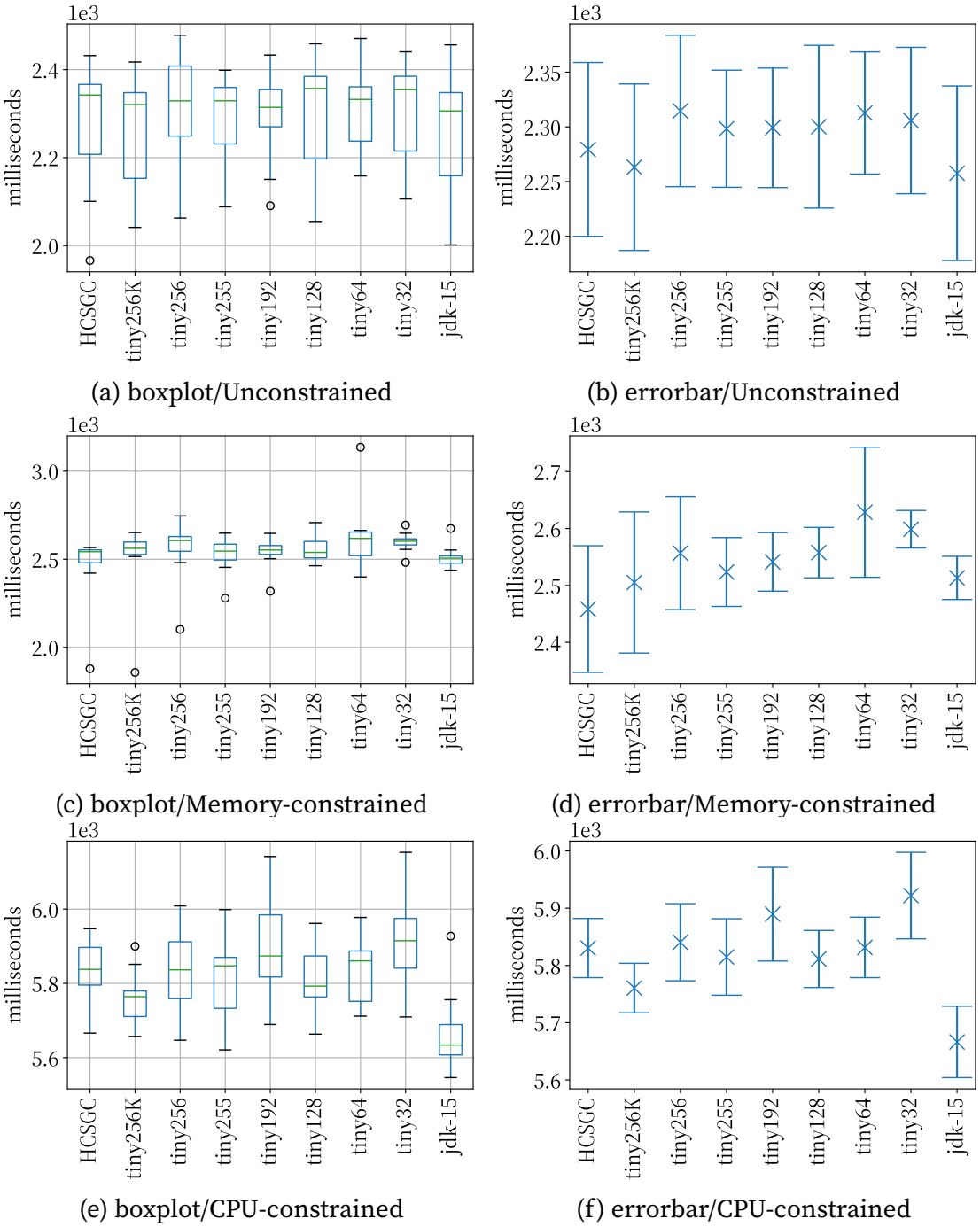
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	2279.39	129.78	5.69%	0.00%	2196.18	2350.34
tiny256K	15	2263.16	127.99	5.66%	0.71%	2181.62	2336.69
tiny256	15	2314.56	114.69	4.96%	-1.54%	2242.48	2379.89
tiny255	15	2298.29	88.40	3.85%	-0.83%	2241.73	2346.55
tiny192	15	2299.17	91.91	4.00%	-0.87%	2240.06	2350.87
tiny128	15	2300.16	125.49	5.46%	-0.91%	2220.91	2369.42
tiny64	15	2312.69	94.01	4.06%	-1.46%	2254.89	2366.82
tiny32	15	2305.81	110.97	4.81%	-1.16%	2235.73	2367.46
jdk-15	15	2257.63	134.60	5.96%	0.95%	2172.84	2333.87

(b) Memory-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	2458.48	208.83	8.49%	0.00%	2321.00	2544.21
tiny256K	15	2505.14	231.94	9.26%	-1.90%	2351.30	2599.17
tiny256	15	2556.71	174.72	6.83%	-4.00%	2443.64	2641.49
tiny255	15	2523.61	102.22	4.05%	-2.65%	2457.32	2576.17
tiny192	15	2541.35	89.54	3.52%	-3.37%	2482.72	2586.36
tiny128	15	2557.63	75.37	2.95%	-4.03%	2516.33	2604.58
tiny64	15	2628.57	197.18	7.50%	-6.92%	2531.66	2759.46
tiny32	15	2598.71	55.90	2.15%	-5.70%	2566.08	2630.74
jdk-15	15	2513.20	65.85	2.62%	-2.23%	2479.79	2556.20

(c) CPU-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	5830.38	89.85	1.54%	0.00%	5776.93	5881.21
tiny256K	15	5760.51	73.58	1.28%	1.20%	5719.64	5805.28
tiny256	15	5840.52	115.64	1.98%	-0.17%	5773.42	5907.35
tiny255	15	5814.81	112.27	1.93%	0.27%	5746.91	5878.92
tiny192	15	5889.50	139.39	2.37%	-1.01%	5810.34	5972.85
tiny128	15	5811.27	86.21	1.48%	0.33%	5761.49	5862.04
tiny64	15	5831.46	90.41	1.55%	-0.02%	5779.37	5885.53
tiny32	15	5922.05	130.55	2.20%	-1.57%	5846.05	5998.03
jdk-15	15	5666.40	108.78	1.92%	2.81%	5610.27	5736.61



(a) boxplot/Unconstrained

(b) errorbar/Unconstrained

(c) boxplot/Memory-constrained

(d) errorbar/Memory-constrained

(e) boxplot/CPU-constrained

(f) errorbar/CPU-constrained

Figure A.7: Benchmark results of lusearch_large

Table A.8: Benchmark results of sunflow_large

(a) Unconstrained

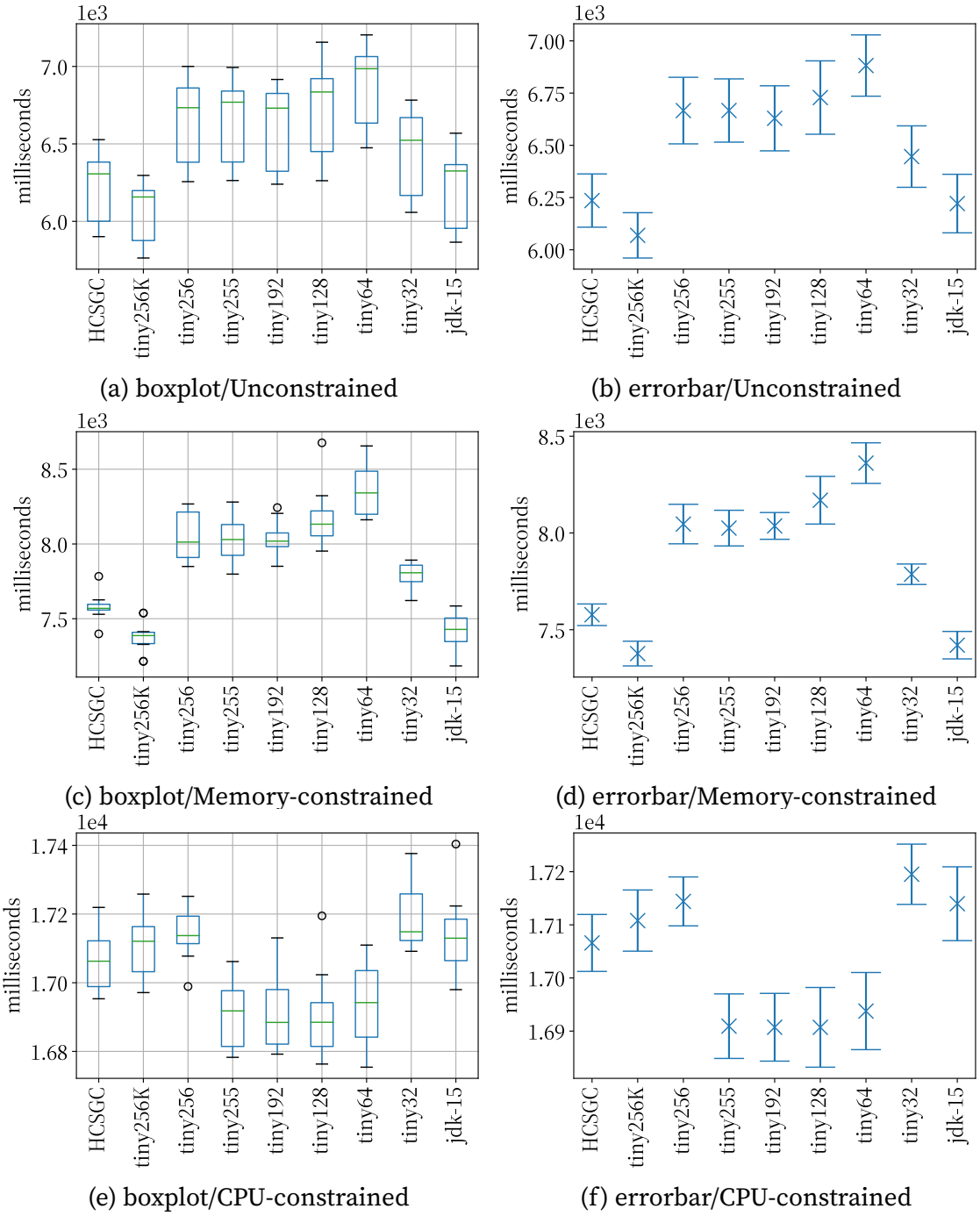
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	6235.58	213.54	3.42%	0.00%	6105.78	6357.86
tiny256K	15	6069.13	181.46	2.99%	2.67%	5960.36	6173.15
tiny256	15	6666.34	269.35	4.04%	-6.91%	6498.80	6823.64
tiny255	15	6666.65	254.42	3.82%	-6.91%	6512.74	6812.64
tiny192	15	6629.27	265.44	4.00%	-6.31%	6466.16	6780.14
tiny128	15	6728.73	295.89	4.40%	-7.91%	6546.48	6895.57
tiny64	15	6881.89	251.48	3.65%	-10.36%	6723.72	7028.29
tiny32	15	6446.16	247.41	3.84%	-3.38%	6295.01	6590.45
jdk-15	15	6221.23	237.29	3.81%	0.23%	6073.47	6356.75

(b) Memory-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	7577.34	94.70	1.25%	0.00%	7521.79	7634.54
tiny256K	15	7376.72	110.25	1.49%	2.65%	7313.01	7441.68
tiny256	15	8045.99	164.34	2.04%	-6.18%	7947.04	8149.68
tiny255	15	8025.04	156.96	1.96%	-5.91%	7931.94	8119.36
tiny192	15	8036.42	118.99	1.48%	-6.06%	7968.72	8107.65
tiny128	15	8169.16	211.83	2.59%	-7.81%	8061.37	8303.47
tiny64	15	8360.92	179.71	2.15%	-10.34%	8260.47	8469.43
tiny32	15	7786.68	89.91	1.15%	-2.76%	7732.91	7836.55
jdk-15	15	7420.15	121.77	1.64%	2.07%	7347.47	7488.85

(c) CPU-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	17065.93	90.87	0.53%	0.00%	17013.01	17120.43
tiny256K	15	17108.02	98.24	0.57%	-0.25%	17049.83	17164.68
tiny256	15	17144.11	77.39	0.45%	-0.46%	17096.89	17187.13
tiny255	15	16909.26	103.94	0.61%	0.92%	16848.92	16969.56
tiny192	15	16907.23	110.82	0.66%	0.93%	16846.80	16976.02
tiny128	15	16907.08	129.76	0.77%	0.93%	16837.25	16987.83
tiny64	15	16937.67	124.11	0.73%	0.75%	16864.48	17010.28
tiny32	15	17195.26	98.10	0.57%	-0.76%	17142.20	17254.68
jdk-15	15	17139.77	119.66	0.70%	-0.43%	17074.68	17216.11



(a) boxplot/Unconstrained

(b) errorbar/Unconstrained

(c) boxplot/Memory-constrained

(d) errorbar/Memory-constrained

(e) boxplot/CPU-constrained

(f) errorbar/CPU-constrained

Figure A.8: Benchmark results of sunflow_large

Table A.9: Benchmark results of xalan_large

(a) Unconstrained

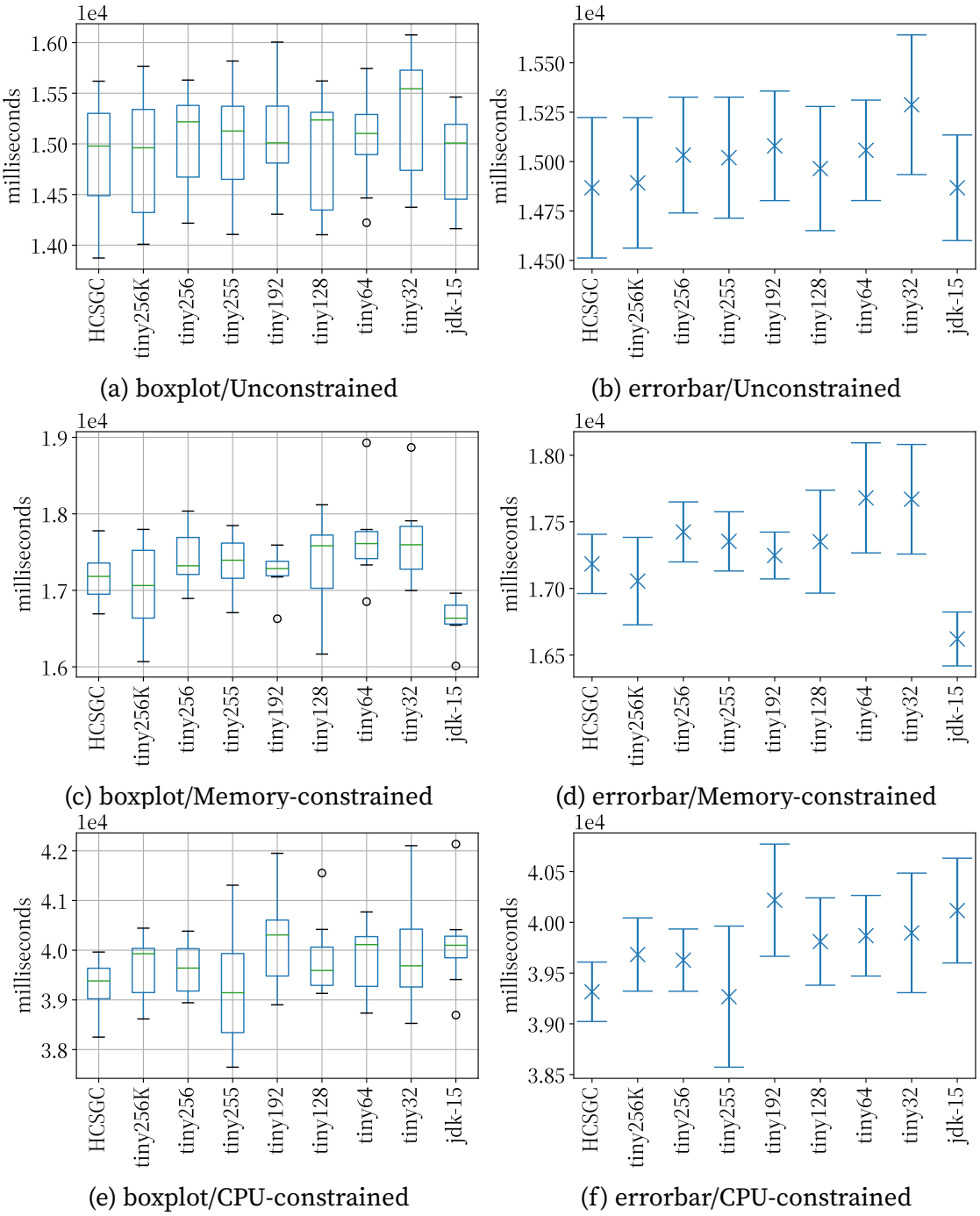
Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	14867.74	564.01	3.79%	0.00%	14508.66	15204.73
tiny256K	15	14892.34	561.47	3.77%	-0.17%	14547.70	15213.06
tiny256	15	15032.89	493.88	3.29%	-1.11%	14728.54	15310.49
tiny255	15	15019.91	515.65	3.43%	-1.02%	14697.24	15310.36
tiny192	15	15079.81	460.62	3.05%	-1.43%	14801.24	15363.23
tiny128	15	14964.86	533.50	3.57%	-0.65%	14636.75	15269.01
tiny64	15	15057.13	416.99	2.77%	-1.27%	14797.92	15301.10
tiny32	15	15287.77	588.10	3.85%	-2.83%	14919.46	15619.54
jdk-15	15	14867.64	449.13	3.02%	0.00%	14594.98	15127.14

(b) Memory-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	17183.80	356.32	2.07%	0.00%	16968.22	17409.31
tiny256K	15	17055.09	563.00	3.30%	0.75%	16723.63	17369.04
tiny256	15	17424.24	369.86	2.12%	-1.40%	17200.29	17669.64
tiny255	15	17353.28	365.88	2.11%	-0.99%	17117.45	17568.11
tiny192	15	17247.15	285.66	1.66%	-0.37%	17053.92	17403.73
tiny128	15	17351.05	632.38	3.64%	-0.97%	16943.00	17715.62
tiny64	15	17680.03	635.10	3.59%	-2.89%	17300.80	18132.79
tiny32	15	17669.64	619.12	3.50%	-2.83%	17294.81	18109.83
jdk-15	15	16620.41	307.48	1.85%	3.28%	16394.79	16797.02

(c) CPU-Constrained

Config	N	Mean	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
HCSGC	15	39315.81	504.89	1.28%	0.00%	39000.59	39590.72
tiny256K	15	39683.28	629.87	1.59%	-0.93%	39291.26	40034.57
tiny256	15	39627.86	523.52	1.32%	-0.79%	39318.57	39933.88
tiny255	15	39267.89	1193.91	3.04%	0.12%	38583.96	39989.11
tiny192	15	40218.55	948.78	2.36%	-2.30%	39670.71	40788.70
tiny128	15	39811.07	739.42	1.86%	-1.26%	39429.45	40302.15
tiny64	15	39868.42	688.59	1.73%	-1.41%	39449.06	40255.63
tiny32	15	39896.46	1012.56	2.54%	-1.48%	39339.35	40529.33
jdk-15	15	40116.31	873.60	2.18%	-2.04%	39644.10	40671.27



(a) boxplot/Unconstrained

(b) errorbar/Unconstrained

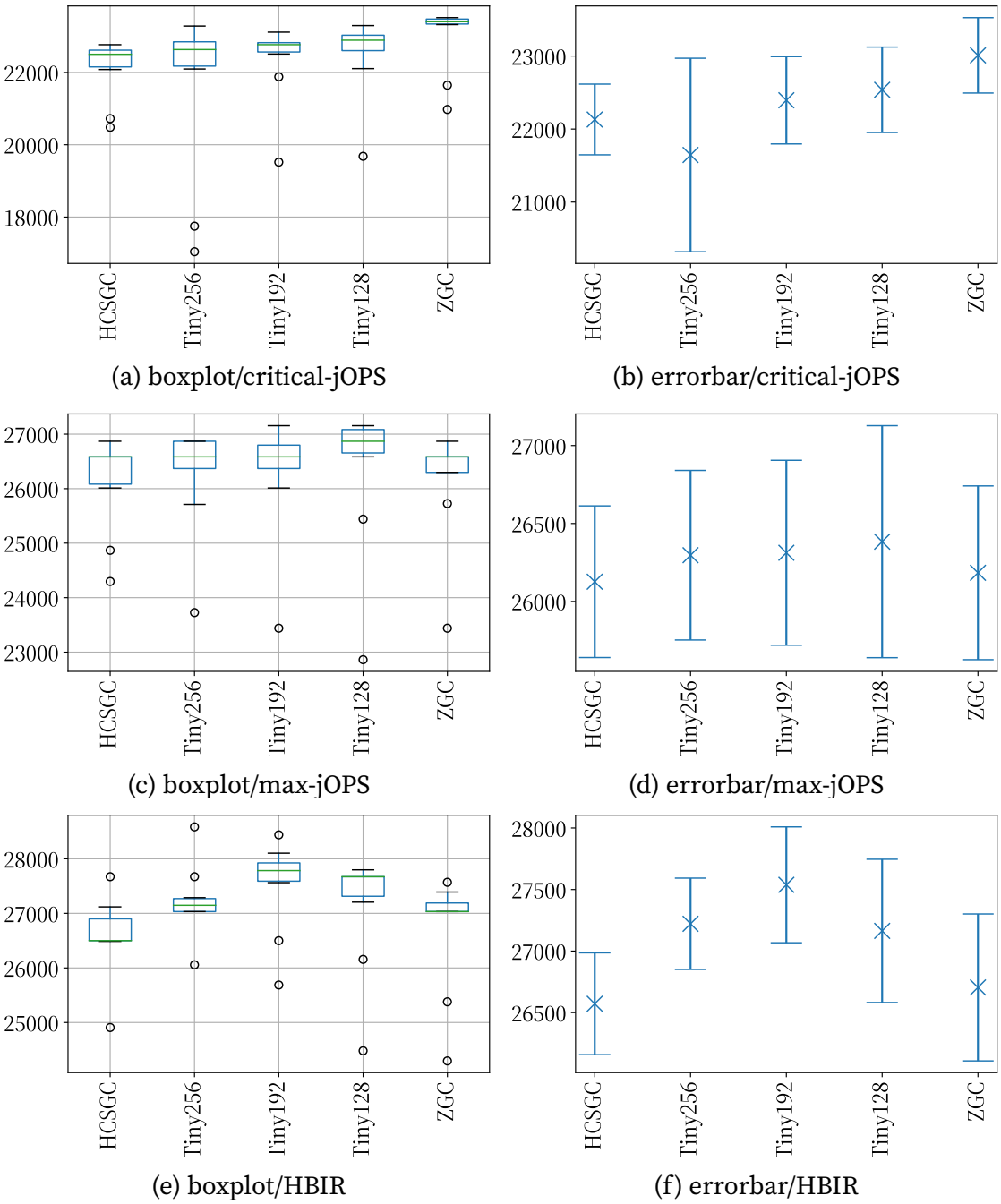
(c) boxplot/Memory-constrained

(d) errorbar/Memory-constrained

(e) boxplot/CPU-constrained

(f) errorbar/CPU-constrained

Figure A.9: Benchmark results of xalan_large



(a) boxplot/critical-jOPS (b) errorbar/critical-jOPS

(c) boxplot/max-jOPS (d) errorbar/max-jOPS

(e) boxplot/HBIR (f) errorbar/HBIR

Figure A.10: Benchmark results of SPECjbb2015

For DIVA

{
”Author1”: {
”Last name”: ”Yu”,
”First name”: ”Jinyu”,
”E-mail”: ”jinyuy@kth.se”,
”ORCiD”: ”0000-0002-3532-2160”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science ”,
}
},
”Degree”: {”Educational program”: ”Master’s Programme, Embedded Systems, 120 credits”},
”Title”: {
”Main title”: ”Improving relocation performance in ZGC by identifying the size of small objects ”,
”Language”: ”eng” },
”Alternative title”: {
”Main title”: ”Förbättrad omplaceringsprestanda i ZGC genom att identifiera storleken på små objekt”,
”Language”: ”swe”
},
”Supervisor1”: {
”Last name”: ”Wrigstad”,
”First name”: ”Tobias”,
”E-mail”: ”tobias.wrigstad@it.uu.se”,
”Other organisation”: ”Uppsala University, Department of Information Technology”}
},
”Supervisor2”: {
”Last name”: ”Lidén”,
”First name”: ”Per”,
”E-mail”: ”per.liden@oracle.com”,
”Other organisation”: ”Oracle”}
},
”Supervisor3”: {
”Last name”: ”Österlund”,
”First name”: ”Erik”,
”E-mail”: ”erik.osterlund@oracle.com”,
”Other organisation”: ”Oracle”}
},
”Supervisor4”: {
”Last name”: ”Thilakasiri”,
”First name”: ”Thilanka”,
”E-mail”: ”thilanka@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science ”,
”L2”: ”Division of electronics and embedded systems” }
},
”Examiner1”: {
”Last name”: ”Becker”,
”First name”: ”Matthias”,
”E-mail”: ”mabecker@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science ”,
”L2”: ”Division of electronics and embedded systems” }
},
”Cooperation”: { ”Partner_name”: ”Oracle”},
”Other information”: {
”Year”: ”2022”, ”Number of pages”: ”xv,99”}
}