JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Jevgēnijs Protopopovs

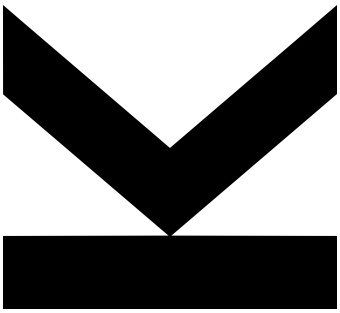
Submitted at
Institute for System Software

Supervisor
o.Univ.-Prof. Dr. Hanspeter Mössenböck

Co-supervisor
Dipl.-Ing. Thomas Schatzl

June 2023

Throughput Barrier Exploration for the Garbage-First Collector



Master's Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

The write barrier used by the Garbage-First (G1) collector is known to negatively impact application throughput. Detrimental effects of the barrier are attributed to its complexity and have been well-known within the OpenJDK development community, with potential impacts quantified in scientific research. This thesis designs two alternative throughput-oriented barriers for G1 and develops necessary garbage collector adaptations. In addition, a prototype of run time dynamic barrier switch is implemented. Proposed throughput-oriented barriers are evaluated using a diverse benchmark suite: it is shown that substantial throughput improvements are achieved at the expense of moderate increases in garbage collection pause times with new throughput-oriented barriers. Trade-offs provided by the new barriers are highly advantageous in certain application classes, although their benefits are not universal due to exhibited pause time impacts. Dynamic barrier switch enables additional flexibility in the context of balance between throughput and pause times.

Contents

List of Listings

List of Figures

List of Tables

Chapter 1

Introduction

This chapter discusses the motivation, the problem statement and proposed approach of this thesis, and provides a short overview of related work.

1.1 Motivation

Java is a widely used programming language, consistently appearing at high places in programming language popularity ratings [72, 9]. The OpenJDK project being the official reference implementation of Java Platform [10]. In accordance with Java Virtual Machine (JVM) Specification [34], OpenJDK implements automatic dynamic memory management via garbage collection. The Garbage-First (G1) collector, introduced by Detlefs et al. [14], is the default garbage collection algorithm since the OpenJDK 9 release [25].

The Garbage-First collector is a mostly concurrent, stop-the-world, generational, and incremental evacuating garbage collector, with a focus on keeping stop-the-world pause time goals [50]. In order to achieve stated goals, G1 executes long-running tasks, such as whole-heap liveness analysis and maintenance of data structures required for incremental evacuation [28], concurrently to the running application.

The G1 collector, being the default collector of OpenJDK project, aims to provide a balance, meeting pause-time goals and achieving high throughput simultaneously [14, 50]. At the same time, OpenJDK also features other garbage collectors targeting different ends of throughput-latency spectrum, such as the Parallel Collector [54] – a garbage collector that focuses on maximum throughput — or ZGC [31, 55] that aims at low latency and high scalability by moving any lengthy operations out of stop-the-world pauses, thus achieving sub-millisecond pause times.

For some applications running under the G1 collector throughput issues were repeatedly reported [65, 64, 20], showing up to 20% worse throughput compared to the Parallel Collector. The write barrier [77] used by G1 to keep garbage collector state in-sync with running application makes large contribution to this throughput difference. For instance, Zhao et al. [76] show that a write barrier similar to the one used by G1 introduces an overhead as high as 12%, whereas shorter and simpler barriers have only moderate overhead of $2 - 6\%$, as shown by Yang et al. [74]. For that reason, proposals [7, 8] have been made to introduce alternative throughput-focused write barriers in G1.

For applications where G1 is capable to perform well within available pause time budget, the current throughput-latency balance is not beneficial, as it prioritizes pause time goals at the expense of throughput. Considering the extent of the throughput deficiency when compared to the Parallel Collector, shifting the trade-off between throughput and

pause times appears feasible. Furthermore, in some applications there are distinct execution phases with different optimal throughput-latency balances. Examples of such software include Web applications and databases, where higher throughput is preferable at start-up phase to achieve quicker transition into the operational phase, and shorter, more predictable pause times are beneficial while serving incoming requests.

1.2 Problem

The Garbage-First collector uses a complex write barrier to track application activities – modifications of object graph on the heap, which are expressed via reference store operations performed by the running program. The write barrier has been present since the introduction of G1 collector in [14], and had evolved within the scope of G1 itself since then [45].

The write barrier is embedded into application code and involves interaction with multiple G1 mechanisms. The barrier encompasses several conditional branches, modifications of global and thread-local data structures, memory synchronization and calls into the G1 collector runtime. Such organization of the write barrier causes throughput regressions in comparison to throughput-oriented collectors — its high complexity and considerable amount of activities result in longer execution times, penalizing every reference modification on the heap. In addition to run time costs, the complex barrier induces pressure during compilation, restricting the range of possible optimizations.

The complex design of the write barrier is the result of requirements imposed by G1 collector properties. Incremental garbage collection requires G1 to track and record cross-references between different heap areas, forcing the barrier to participate in card table and remembered set maintenance. Concurrent operation has G1 ensure that interaction between application and collector threads is correct from a memory consistency perspective, adding memory barrier instructions. The write barrier also performs pre-filtering of references to skip expensive steps of the barrier itself and reduce work done later at collection time.

G1 relies upon guarantees provided by the write barrier, thus changes to the barrier have to be accompanied with updates to the garbage collector. These changes need to preserve its core properties, such as concurrent liveness analysis and incremental collections: in presence of alternative garbage collectors, major changes in throughput-pause time balance are not desired.

While exclusive use of throughput-focused write barriers might be beneficial for some workloads, there exists an already mentioned class of applications, where the optimal set of trade-offs changes throughout the application run time. For such software neither current, nor alternative write barriers might be fully satisfactory, therefore dynamic barrier switch depending on application run time state becomes viable. Such technique creates a new challenge of implementing a mechanism to seamlessly switch barriers and adjust garbage collector state. The extent to which current architecture of HotSpot virtual machine is capable to support dynamic barrier switch is unclear prior to implementation. Furthermore, dynamically switched barrier is expected to present a unique trade-off between throughput and pause times.

1.3 Proposed solution

Discussion on potential changes to the G1 write barrier had already been started in the OpenJDK development community [7, 8]. The main idea of the proposals is a drastic simplification of the write barrier and corresponding adjustment of affected G1 collector mechanisms: current barrier implementation provides certain guarantees, and changing the barrier in isolation would break assumptions made in other parts of the garbage collector. An important aspect of the barrier change proposals is preservation of G1 garbage collection cycle and its main properties, such as concurrent liveness analysis and avoidance of full garbage collections — changes to the garbage collector need to be very focused and limited in scope to avoid major shifts in G1 behavior.

This thesis follows the line of thought expressed in the mentioned proposals and expands on it. Namely, after examining the write barrier, this thesis formulates two throughput-oriented barrier variants of decreasing complexity by stripping the original G1 write barrier of the most computationally-expensive operations. The first variant adapts current G1 write barrier by removing concurrent remembered set maintenance, but keeping the rest of the barrier intact. The second variant also removes pre-filtering, further reducing remembered set maintenance portion of the barrier to a minimal card-marking barrier, similar to the one used by the Parallel Collector [43]. Both barrier variants still preserve parts that assist in concurrent liveness analysis, and adaptations of the G1 collector are limited to changes in card table and remembered set maintenance, which introduce extra work into garbage collection pause. This thesis discusses several potential alternative garbage collector adaptations which differ in their degree of divergence from current G1 behavior. Throughput write barrier variants and garbage collector adaptations are validated for correctness with applicable tests from OpenJDK test suite.

A diverse set of benchmarks is composed to examine newly defined write barrier variants, their impact on application throughput and garbage collection pause times. This benchmark suite includes both well-known JVM performance benchmarks and performance-oriented workloads, which were previously used to demonstrate deficiencies in G1 throughput. Benchmarks are primarily focused on measuring impact of barrier changes on application-level behavior. Effects of new write barriers include higher application throughput due to more lightweight barriers, as well as longer garbage collection pauses due to extra work performed at pause time.

This thesis also proposes a prototype of dynamically switched barrier based on deoptimization. The implemented policy includes both an API to let the application switch barriers programmatically, as well as a heuristic which makes decisions to switch based on anticipated pause time impact on the application. The dynamically switched barrier is validated and evaluated using the same benchmarks and test suite.

1.4 Related work

This section presents a short overview of prior work related to garbage collector barrier performance, and performance of G1 collector particularly.

Discussion on the impact of garbage collector barriers had started along with introduction of generational and incremental garbage collection techniques. Such schemes may perform collections on the subset of the whole heap, therefore identification of live objects and cross-references is crucial for establishing correctness of these algorithms, and barriers are used to ensure that. Zorn [77] performs an early analysis of different kinds of barriers,

including hardware- and OS-assisted barriers, and concludes that software-defined write barrier overheads can be as low as $2 - 6\%$; barrier impact is being analyzed in isolation. Another early work detailing specifically on the impact of different write barriers was conducted by Hosking et al. [23], who analyzed the performance costs in context of full application run time. That study also confirms feasibility of software-defined write barrier and shows that different barrier variants have impact on the application-level performance. A card-marking barrier is introduced by Wilson et al. [73] and further optimized by Hölzle [22]. Although, these early works provide valuable historical insight on the garbage collector barrier impact analysis, their utility in the context of this thesis is limited: these studies had been conducted prior to public releases [56] of both Java language itself and HotSpot VM, a highly optimizing JVM implementation used by OpenJDK project [41], therefore their results may not be fully applicable to modern Java applications. Besides that, the methodology used in these studies may not fully align with performance analysis approaches that take into account specifics of JVM environment and are discussed later in the section.

The Garbage-First collector had been introduced by Detlefs et al. [14]. The original design described in the paper shows that certain compromise in terms of throughput had been made in order to achieve soft real-time goals. However, the authors did not provide detailed analysis of trade-offs and possible optimizations in that respect, leaving write barrier improvements for future work. Furthermore, the G1 collector has constantly evolved since it's inception, so conclusions made in the original paper might need to be revisited now. A recent attempt to evaluate different aspects of G1 had been made by Zhao et al. [76]. In that paper the authors perform assessment of different G1 collector mechanisms in isolation, and conclude that the impact of G1 write barrier might be as high as 12%. Even though the results were based on re-construction of G1 design on top of Jikes RVM [2] and do not use OpenJDK G1 implementation directly, the work still provides valuable observations on potential costs of the G1 write barrier.

A considerable number of studies have investigated performance impact of garbage collector barriers in isolation for Java. Blackburn et al. [5] show that the impact of a reasonable generational write barrier is moderate – less than 2% on average and not more than 6% in the worst case, yet the exact results are architecture-specific. A more recent paper by Yang et al. [74] revisits and confirms these conclusions. The methodology used by these studies, however, is not compatible with aims of this thesis – the authors perform focused analysis of barrier impact in isolation, analyzed barriers do not resemble the write barrier used by the G1 collector, and the impact on pause times and application-level performance is ignored. Blackburn et al. [4] also discuss the impact of barrier inlining on performance, showing that correct separation between the fast- and slow-paths is crucial for both runtime and compilation performance. Such observation can be particularly topical for complex write barriers, such as the one used by the G1 collector. Even though the methodology used in these studies might not be directly applicable in the scope of this thesis, their conclusions suggest that a simpler well thought out write barrier is very moderate in terms of performance costs.

A discussion on other potential approaches of improving G1 collector throughput has happened in the OpenJDK development community. One of the proposals [63] demonstrates deficiencies in current region allocation algorithm and suggests potential improvements in that regard. Another avenue of improving the throughput is rearranging certain activities to be run concurrently to the mutator [67]. Stronger guarantees in object allocation may enable barrier elision [66], which is also beneficial for throughput. Many of proposed improvements are not mutually exclusive with the solution proposed in this

thesis, thus can be implemented alongside with it. Furthermore, an alternative approach to the write barrier optimization has also been discussed [68] – the proposal preserves collaboration with concurrent collector activities within the write barrier, but replaces an expensive memory barrier instruction with a different synchronization technique, thus producing shorter write barrier. By preserving the concurrent activity support, that proposal differs from the solution attempted in this thesis, however it demonstrates the diversity of viable throughput improvement ideas.

Finally, an extensive body of Java performance benchmarks has been developed over the years. These include benchmarks developed by *The Standard Performance Evaluation Corporation (SPEC)*: SPECjbb2005 [11, 1] and SPECjvm2008 [12], as well as their variants, such as pjbb2005 [3]. Other notable and widely used benchmarks are DaCapo [6] and Renaissance [58]. Lengauer et al. [30] study benchmark selection methodology in accordance with application characteristics. Georges et al. [18] define a statistically rigorous methodology of performance evaluation. These benchmarks and measurement methodologies delineate a set of best practices that are useful in the evaluation part of this thesis.

1.5 Structure of the thesis

Chapter 2 contains background information on the important aspects of OpenJDK and the Java Virtual Machine, Garbage Collection theory, as well as a detailed description of relevant Garbage-First collector concepts and the analysis of the problem.

Chapter 3 presents several throughput-oriented barrier variants for G1, examines adaptations in G1 necessary for the barriers to work, discusses possible configurations of the adapted G1 collector, and introduces a dynamically-switched barrier and challenges concerning its implementation.

Chapter 4 defines a performance measurement methodology used for the evaluation of changes, introduces a benchmarking suite, describes hardware setup used for benchmarking, presents the benchmarking results and their interpretation. Finally, a heuristic for a dynamically-switched barrier variant is devised and evaluated based on performed analysis.

Finally, chapter 5 summarizes the results and concludes the thesis, discussing prospects of future work on the topic.

Chapter 2

Background

This chapter presents essential concepts relevant in the context of this thesis. The chapter begins with a brief description of OpenJDK and the Java Virtual Machine, which is followed by an explanation of fundamental aspects of garbage collection and their implementation within the Garbage-First collector. Finally, a more detailed reformulation of the thesis problem and its analysis is provided.

2.1 OpenJDK & Java Virtual Machine

The OpenJDK project provides an open-source reference implementation of the Java Platform, Standard Edition [10], in accordance with the Java Virtual Machine Specification [34] and the Java Language Specification [19]. The Java Virtual Machine Specification defines an abstract computing machine, capable of executing bytecode in a portable manner. The Java Virtual Machine has no direct dependency on the semantics of Java programming language, and therefore can host other programming languages as well. The Java Development Kit (JDK) developed by OpenJDK [40] is the reference implementation of these specifications and includes components, such as HotSpot virtual machine, javac compiler and Java Class Library.

The HotSpot virtual machine is a highly-optimizing cross-platform implementation of Java Virtual Machine. It implements an interpreter and an adaptive compiler [52] to deliver optimal performance of executed code. There are two Just-in-Time compilers [37]:

- C1 — a lightly-optimizing fast bytecode compiler. Implements basic optimization and code generation techniques.
- C2 — a highly-optimizing bytecode compiler. Provides a wide range of sophisticated optimizations and advanced code generation facilities.

The HotSpot virtual machine performs runtime analysis to identify performance-critical code paths and compile these parts with an increasing level of optimization [38]. Infrequently used code is interpreted via the means of a template interpreter. The template interpreter relies on a table of machine-specific assembly code fragments, which specify implementations of each bytecode. Overall, HotSpot heavily depends on machine code generation to implement both the template interpreter and the Just-in-Time compilers, therefore its source tree is comprised of generic (*shared*) and platform-specific (*cpu* and *os*) parts.

In accordance with the Java Virtual Machine specification, HotSpot implements automatic memory management of the Java application heap using garbage collection. The specification requires automatic storage management with no explicit object deallocations, however it does not impose any further constraints on the garbage collection techniques used by the implementation. HotSpot defines a generic garbage collector interface [27, 44], enabling different algorithms of garbage collection that can co-exist within HotSpot code base. Any available collector can be selected and configured by the end-user at virtual machine start-up. As of JDK 20, HotSpot provides following garbage collectors [42]:

- Serial Collector — single-threaded garbage collector, targeted for single processor machines and applications with small data sets.
- Parallel Collector (also called the Throughput Collector) — multi-threaded generational collector, targeted for multiprocessor systems and applications with medium and large data sets.
- Garbage-First Collector — a scalable mostly concurrent collector, targeted at a wide range of configurations, designed to meet pause time goals and provide high throughput simultaneously. The default garbage collector.
- Z Garbage Collector — scalable concurrent garbage collector focused on low latency. Designed to sacrifice throughput in exchange for a few millisecond long collection pauses.
- Shenandoah — a concurrent garbage collector focused on consistent, heap size independent pause times [17].
- Epsilon — a passive garbage collector that does not implement any memory reclamation. Shuts down the Java Virtual Machine in case of heap exhaustion. Targets lowest possible latency overhead [70].

2.2 Garbage collection

Garbage collection is an approach to automatic dynamic memory management that involves automatic discovery and reclamation of memory that is no longer in use by a program. Runtime systems with garbage collection feature a semi-independent component — the garbage collector — that possesses global knowledge about dynamically allocated objects and their relationships, and is capable of unreachable object identification and reclamation. The heap all garbage collectors in HotSpot operate on is a contiguous range of memory where objects are allocated by the running program. The objects form an object graph by creating references between objects. With respect to the garbage collector, the running program is called mutator. In the context of garbage collection in HotSpot, relevant activities of the mutator are object allocation and changes to object references. These changes may be applied to the reference fields contained in the objects on the heap, as well as reference fields contained in other locations — thread stacks, static variables, etc. Object reachability analysis (tracing) is performed with respect to particular reference fields, the roots. The garbage collector typically has exclusive rights to deallocate objects, thus eliminating such problem classes as dangling pointers, use-after-free and double-free issues [26]. As a memory management technique, garbage collection has been first introduced by McCarthy [35].

Throughout time multitude of garbage collection algorithms were introduced and implemented. Based on a comprehensive study of known approaches performed by Jones et al. [26], a brief description of concepts and their implications, relevant within the scope of this thesis, is provided below:

2.2.1 Generational & Incremental garbage collection

Based on the weak generational hypothesis which states that most objects survive only for a short period of time, generational garbage collectors segregate heap into distinct areas called generations [32]. Objects are placed within and promoted between generations based on their age — number of garbage collections the object survived. Garbage collections of the young generation are performed independently, thus yielding the most benefit from reclaiming shortly-lived objects, whereas whole heap collections are infrequent. However, separate collections of different generations introduce a need to keep track of inter-generational references in order to perform object reachability analysis correctly, therefore imposing certain bookkeeping overhead.

Generational garbage collectors are a subclass of incremental collectors. With the incremental collection approach, the heap is partitioned into a set of distinct regions, and only a subset of all heap regions are collected at once. Exact heap layout and principles of selecting the heap region subset for each collection (the collection set) might follow generational hypothesis (as for generational collectors) or some other considerations, depending on the collector implementation.

2.2.2 Remembered sets

In general, any reference that references an object outside of its own heap region is of interest for incremental garbage collectors. When collecting an increment, the garbage collector requires knowledge of all references into that increment. Full heap tracing to discover them is unfeasible in such type of collectors, therefore these *"interesting" reference locations* from heap regions outside of the collection set need to get treated as heap roots — locations where liveness analysis starts from — even if the containing object might be unreachable itself. Thus, locations of "interesting" references created by the mutator between garbage collections need to get recorded.

Remembered sets are data structures used to record these references of "interest" for the garbage collector. Remembered sets hold reference source locations — typically objects or object fields — with varying granularity. In order to identify references into the collection increment, the garbage collector only needs to inspect locations stored in the remembered sets. Remembered sets are frequently implemented using hash tables and sequential store buffers. The sequential store buffer may act as front-end for hash tables, thus batching expensive hash table update operations. Furthermore, in multi-threaded environments thread-local remembered set data structures eliminate synchronization and contention on global data structures.

The main benefit of remembered set approach is precision and reduction of overhead during garbage collection. Garbage collector needs to do little scanning beyond extraction of references from remembered sets: a remembered reference often is directly treated as a heap root. The main drawback is a dynamic nature of remembered set data structures. If sequential store buffer gets full, it needs to be processed in order to guarantee enough space for incoming references, thus performance of remembered sets depends on modification frequency.

Implementations of remembered sets vary greatly in terms of underlying data structures, granularity and trade-offs. For more details refer to the respective sections of [26].

2.2.3 Card table

The card table is a coarse representation of the whole heap where each card corresponds to a distinct relatively small area (usually not larger than 512 bytes) on the heap. The value of a card indicates whether respective heap area may contain "interesting" references. Heap areas that may contain such references are subsequently scanned by the garbage collector. A card that indicates need of scanning is commonly called *dirty*. The mutator is responsible for card marking as part of reference field update. HotSpot represents the card table as a fixed array of bytes.

The main benefit of card marking approach is simplicity and efficiency of dirtying a card: calculation of the card address and modification of the card table consists only of a few machine instructions with no branching. Furthermore, the card table is a fixed data structure which cannot overflow and its behavior is independent of modification frequency. The main drawback of the card table is imprecision and collection time overhead. For each dirty card, the garbage collector needs to scan the whole corresponding range of the heap. Moreover, in multiprocessor systems unconditional card marking schemes are known to introduce cache contention [15], potentially degrading the throughput.

The card marking approach was introduced by Sobalvarro [71] and Wilson et al. [73]. Possible optimizations to card marking scheme were discussed by Hölzle [22].

Another optimization to avoid full card table scans during the garbage collection pause are two-level card tables. Higher level cards represent ranges of cards from the lower levels, dirty values indicate whether there are dirty cards in the respective range.

2.2.4 Barriers

Barriers are part of the garbage collector runtime interface which defines interaction between the mutator and the collector. A barrier is a fragment of memory management code that is embedded within the mutator code and executed whenever an operation requires coordination between mutator and garbage collector. For example, in incremental garbage collection algorithms the mutator typically needs to identify references that cross heap region boundaries, so a barrier is executed whenever an operation involving heap references occurs in the running program. Concurrent garbage collectors — those that perform certain garbage collection stages, including object graph tracing and object evacuation, concurrently to the running mutator — also use barriers to establish consistency between mutator and collector activities

Barriers are classified with respect to the type of the operation on the object reference into read and write barriers. The latter are particularly important in the context of this thesis. These write barriers track mutator modifications to the object graph on the heap. Whenever a reference field in a heap object gets updated by the mutator, the write barrier inspects involved references, identifies the "interesting" ones and updates the garbage collector with that information. For example the card table and remembered sets need to be updated to hold identified references. This way, on each incremental collection the garbage collector is able to identify the changes introduced to the heap by the mutator and apply that knowledge to correctly find reachable objects. Write barriers often include pre-filtering conditions, and barrier code is split into fast and slow paths with respect to those. The split between fast and slow path, as well as decisions on

inlining respective code fragments depend on anticipated frequency of different reference classes and performance impact of the barrier.

Design of a write barrier depends on the garbage collection algorithm, involved data structures and expected frequency of reference writes. Different forms of barriers, as well as other reference identification mechanisms, are discussed by Zorn [77] and Hosking et al. [23]. Blackburn et al. [5] and Yang et al. [74] study the impact of read and write barriers. Blackburn et al. [4] demonstrates the effects of barrier inlining decisions. The locality of different barriers is studied by Hellyer et al. [21]. Finally, Jones et al. [26] puts barriers into a greater perspective of garbage collection.

2.3 Garbage-First collector

The Garbage-First [14] collector is a generational, incremental, stop-the-world, mostly concurrent evacuating garbage collector, designed to reach both high throughput and short collection pauses at the same time. G1 operates on uniformly-sized regions of the heap, applying a set of heuristics to guide its behavior. The collector carries out evacuation during stop-the-world pauses. Whole-heap activities, where amount of work is proportional to the heap size, are performed mostly concurrently to the mutator. A summary of G1 design is presented below based on analysis of the collector implementation provided in OpenJDK 20 [40] and respective documentation [50]. The collector has been in active development since its inception, therefore the description in the original paper does not reflect its current state.

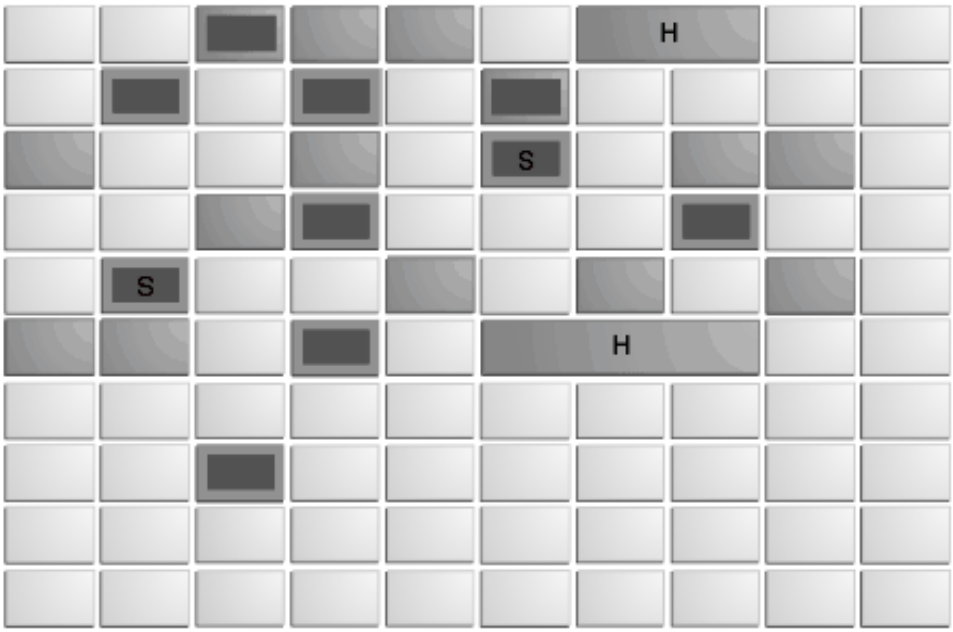


Figure 1: Garbage-First collector heap layout [50]

The Garbage-First collector splits the heap into a set of equally-sized regions, where each region is a distinct continuous area of heap memory whose size is power of two. Figure 1 demonstrates an example heap layout, organized into uniformly-sized regions, where active heap regions are shown with a dark background color. The G1 collector follows the generational hypothesis by dividing heap regions into young and old generations, which is reflected by the type of a region: young generation (marked with smaller darker

rectangles on the figure) consists of regions of *eden* and *survivor* types (marked with `S` on the figure), whereas old generation consists of *old*, *archive* regions (containing immutable content, marked as *archive*) and *humongous* regions. Humongous regions are intended to store particularly large objects, spanning multiple heap regions.

Allocation of objects happens on a per-region basis, and, with the exception of humongous objects, new objects are allocated in the eden regions. During subsequent garbage collections objects from eden are evacuated into to survivor regions. Objects that stay alive long enough are promoted into the old generation. Region placement in the heap is not contiguous with respect to the region type. In addition to the type, each region has multitude of other attributes. G1 is an evacuating garbage collector, therefore it copies live objects to reclaim space.

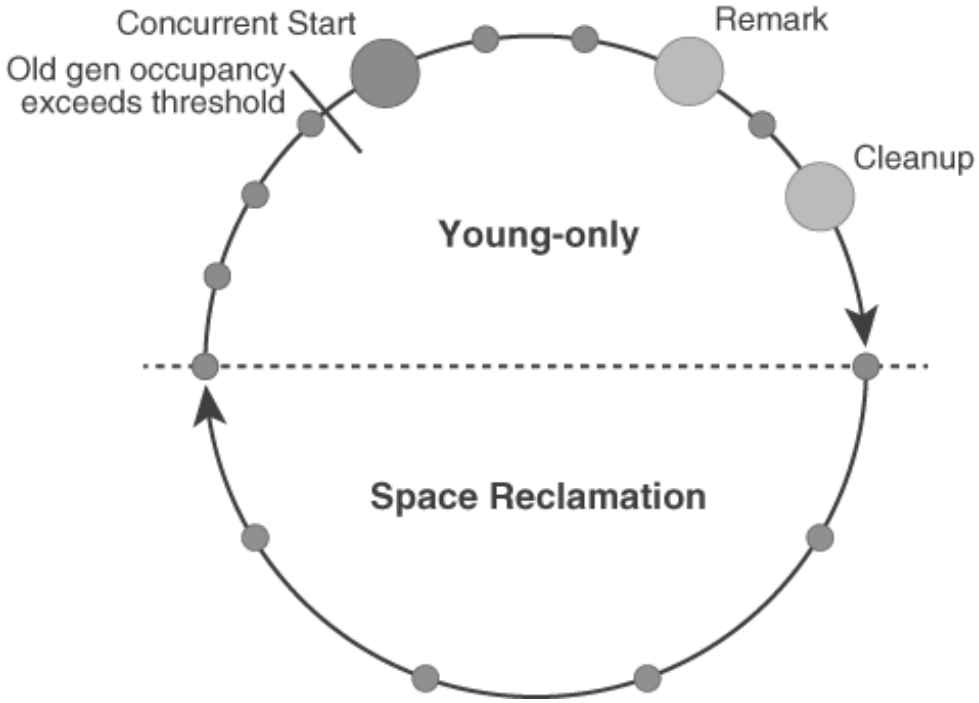


Figure 2: Garbage collection cycle overview [50]

G1 performs collections incrementally by selecting a subset of regions for each collection (the *collection set*). Regardless of the collection type, young generation regions are always included into the collection set. The inclusion of old generation regions depends on the collection type, and the set of collected old regions are determined heuristically. Furthermore, G1 might also form an optional collection set whose evacuation depends on the available pause time budget. Consecutive incremental collection pauses interleaved with periods of the mutator activity make up a garbage collection cycle, which is depicted on the figure 2 and consists of two phases:

- *Young-only phase* encompasses garbage collections that operate exclusively on the young generation regions, thus incrementally promoting surviving objects into the old generation. After the old generation is filled up to a particular threshold, the G1 starts preparing for the next phase by initiating concurrent marking, which also introduces several special pauses for concurrent start, remark and cleanup phases.
- Once concurrent marking finishes, G1 might transit into the *space reclamation phase*. During that phase G1 carries out mixed collections, where collection set also includes

old generation regions. Once enough space has been reclaimed, G1 transitions back into the young-only phase, thus closing the cycle.

G1 is also capable of doing full collections, which perform object reachability analysis and evacuation on the whole heap during a single stop-the-world pause. Full collections might be triggered programmatically by the running application, as well as occur when incremental collection cycle fails to reclaim enough space for normal operation. The latter case is considered to be a sign of inadequate garbage collector setup.

Object reachability analysis is performed concurrently to the running mutator using a *snapshot-at-the-beginning (SATB)* [75] concurrent marking algorithm. The algorithm identifies objects which were unreachable at the beginning of the marking. Newly allocated objects are considered alive in the context of on-going concurrent marking. In order to preserve this SATB invariant, the mutator needs to save references to heap objects it overwrites when concurrent marking occurs. G1 uses a write barrier and per-thread marking buffers for that purpose. Despite being mostly concurrent process, marking includes a few short pauses that are used to initialize marking structures, complete the marking process and prepare for the space reclamation phase of garbage collection cycle respectively. As a result of concurrent marking, the collector obtains information on object reachability and heap region occupancy, and uses it to select old generation region candidates for collection and rebuilding remembered sets. Regions containing only garbage are reclaimed as part of the process. Schatzl [62] provides a detailed description of concurrent marking.

As an incremental, evacuating garbage collector, tracking inter-region references is crucial for correct operation. G1 determines live objects within the regions in the collection set without re-doing full heap analysis on every collection. This requires tracking and update of all references across the heap pointing to objects it copies during the evacuation. G1 stores complete and up-to-date information on the cross-region references on a per-region basis in the remembered sets and the card table data structures. Both data structures have granularity of a card, which represents a 512 byte large area of the heap. Each remembered set contains cards that indicate potential locations of references into the corresponding region. The remembered set data structure is based on a hash table that stores per-region card sets and a list of embedded references from compiled code. The card table covers the whole heap and is used as an intermediate remembered set store [61]:

- During mutator execution, as part of the write barrier — creation of cross-region references by the mutator is marked by dirtying corresponding cards on the card table and enqueueing the cards for further processing to be sorted into relevant remembered sets. The process of scanning a newly dirtied card and updating remembered sets is called refinement, and is normally performed concurrently to the mutator. A detailed description of these mechanisms is available in sections 2.3.1 and 2.3.2.

- As part of evacuation pause — remembered sets of collection set regions, along with other sources of dirty cards, are merged back onto the card table, thus obtaining a unified, de-duplicated remembered set for the collection set regions. Locations that correspond to dirty cards are subsequently scanned in order to identify actual cross-region references. In order to avoid full card table scan at pause time, G1 uses a multi-level card table. Detailed description of this process is available in the section 2.3.3.

G1 extends the amount of possible values of cards on the card table compared to traditional use: besides dirty and clean cards, it defines also “young gen.” cards — those that

represent memory ranges contained in the young generation regions, as well as "already scanned" cards during garbage collection — those that have already been processed. Additional card values help in reducing the number of card table modifications at mutator time and card scans during collections.

G1 design and implementation take specifics of multi-threaded execution environment into account. Write barriers use thread-local data structures in order to batch notifications to the garbage collector, avoiding contention on global data structures. Furthermore, due to portions of garbage collector running concurrently to the mutator, memory barrier instructions are used to ensure consistency. During garbage collection pause, G1 utilizes available resources by using multiple parallel worker threads to perform certain tasks.

Throughout different stages of garbage collection, G1 measures and records various kinds of runtime statistics, including lengths of different collection stages, allocation and card dirtying rates, and others. These statistics are used to estimate costs of future activities and adjust the behavior accordingly. Different G1 aspects, such as collection set selection, young generation sizing, concurrent marking start, are guided by the heuristics in order to achieve predictable behavior.

The Garbage-First collector also includes verification mechanisms that, once enabled through the command line flags, introduce pre- and post-collection checks that inspect current garbage collector state, detecting inconsistencies in broken invariants. The verification is targeted at the garbage collector developers, enabling early detection of issues.

2.3.1 Barriers

The Garbage-First collector uses a write barrier to record mutator changes to the object graph on the heap. The write barrier is implemented for the HotSpot interpreter, C1 and C2 JIT compilers separately, thus the actually executed barrier code depends on the optimization level, however all write barrier implementations share the same logic. The write barrier is executed for all reference write operations, and as part of pre-generated code fragments (stubs) for array copying, checked casts and some others. The optimizing compiler is able to omit the write barrier completely or partially in certain cases.

The write barrier of G1 collector is structured with respect to the reference write operation into two independent parts, serving different purposes:

- *Pre-write* barrier — is executed prior to the reference write and is used to record the previous value of the write target field. The pre-write barrier maintains the snapshot-at-the-beginning invariant of conditional marking. It conditionally filters "interesting" references — non-null pointers when conditional marking is on-going — and records those into per-thread conditional marking buffer. In cases, when the buffer is full, the pre-write barrier invokes the runtime to perform processing. The pre-write barrier is out of scope for this thesis, therefore no further details are provided.

- *Post-write* barrier — is executed after the reference write and is used to record locations of newly introduced cross-region references. The post-write barrier maintains remembered sets. Current implementation of post-write barrier portion encompasses several checks, interaction with the card table and dirty card queues. The rest of this section discusses post-write barrier implementation in detail.

```
1 final CardTableBase = /* Offset between heap start and card table start */;
2 final RegionLogSize, CardLogSize = /* Logarithms(2) of heap region and card size */;
```

```
enum Card {
    Dirty,
    Young
}

public static void postWrite(Location field, Reference newValue) {
    // 1. Check whether reference crosses region boundaries
    if ((field ^ newValue) >> RegionLogSize == 0)
        return;

    // 2. Check whether new value is not null
    if (newValue == null)
        return;

    // 3. Calculate card address
    CardAddress card = CardTableBase + (field >> CardLogSize);

    // 4. Check whether the store region is not young
    if (card.load() == Card.Young)
        return;

    // 5. Memory barrier to ensure access consistency
    MemoryBarrier.storeLoad();

    // 6. Check whether the card is not already dirty
    if (card.load() != Card.Dirty)
        return;

    // 7. Dirty the card
    card.store(Card.Dirty);

    // 8. Enqueue the card for refinement
    DirtyCardQueue queue = ThreadLocalData.dirtyCardQueue();
    if (!queue.tryEnqueue(card))
        G1Runtime.enqueueDirtyCard(card);
}
```

Listing 1: Pseudocode of G1 post-write barrier

Listing 1 shows pseudocode for the post-write barrier used in JDK 20 [45, 46, 47] algorithm. It is structured as follows:

1. Cross-region store check — the newly introduced reference must cross region boundaries in order to be "interesting" for the garbage collector. As heap region size is guaranteed to be power of two, addresses of objects from the same region always have a common prefix, unique for each region, and differ only by an offset from the region base address. The implementation in G1 optimizes this check by using exclusive-OR operation to combine referenced object and store location into a single value and using bit shift to remove the offset part of the address, thus producing zero if objects are located within the same region. In such case, the reference store is not "interesting" and the rest of the barrier can be skipped.

2. Null pointer check — stores of null pointers are not "interesting" for the garbage collector, as they do not introduce cross-region references. The respective check is trivial and G1 includes it as the second condition for the barrier.

3. Card address calculation — the post-write barrier uses card table as an intermediate storage for identified locations of "interesting" references. The card number is calculated by bit shifting the store target field address. The resulting value, however, is not a correct card address, as it assumes the beginning of process memory space as card table base. The code adds a pre-computed offset to the bit shift result in order to obtain valid card address within the card table.

4. Young generation check — young generation regions are always included into the collection set, therefore all references located in the young generation will be scanned regardless of remembered set contents. Thus references originating from the young generation regions are not "interesting" to garbage collector and can be skipped by the barrier. Young generation cards are specially marked on the card table.

5. Memory barrier — the concurrent nature of refinement requires synchronization between mutator and refinement threads. The write barrier includes a store-load memory barrier to ensure that effects of the reference store operation are visible to other threads prior [29] to conditional card dirtying. Failure to establish correct memory operation order can lead to a situation when current card value is loaded before the reference store operation. If the card is being refined at the same time, it might get cleared and refined prior to the effect of reference store becomes visible, which leads to incompleteness of the card table state and violates correctness of the garbage collector.

6. Dirty card check — cards that are already marked dirty require no further actions for the barrier, because they are already pending for refinement. In that case, the rest of the barrier can be skipped.

7. Card dirtying — card value on the card table is being modified in order to indicate that the respective memory range contains "interesting" references. At this point, all checks have already been done and irrelevant references were filtered out.

8. Card enqueue for refinement — dirty cards from the card table need to get processed by concurrent refinement mechanism in order to maintain remembered sets. The detailed description of refinement is given in the section 2.3.2. The write barrier uses per-thread dirty card queue (sequential store buffer) to record newly dirtied cards. In case the buffer is full, the garbage collector runtime is invoked to process the buffer, and possibly perform self-refinement.

As a result of the post-write barrier, "interesting" reference stores are filtered from the rest, marked on the card table and enqueued for remembered set maintenance. The post-write barrier has significantly evolved since the original version was introduced by Detlefs et al. [14]: the original version only included fast-path checks and invoked a runtime subroutine to perform actual card table and dirty card queue ("remembered set log") modification, whereas current version includes most of work into the barrier itself, introduces extra checks and relies on the runtime code only in case of dirty card queue overflow.

The exact code of the post-write barrier depends on the CPU architecture and code optimization level. The template interpreter and runtime stubs implement described barrier algorithm in a straightforward manner [45]. The C1 compiler splits the barrier into fast-path — cross-region and null pointer checks — and slow path [46]. The C2 compiler uses versatile SSA representation to implement barrier [47], and exact form of

barrier, use of machine instructions and inlining decisions are made by the optimizing compiler. Figure 3 depicts the post-write barrier assembly code generated for x86_64 architecture by the C2 compiler; an assembly listing is available in the appendix A.1. The code is decomposed into fragments that correspond to algorithm steps, fragments are delimited on the figure, showing complexity of each step. Barrier fragments are interleaved with other code, thus splitting it into inline fast-path and out-of-line slow-path parts.

```
da3b:    xorq    %r11, %r10
da3e:    shrq    $0x15, %r10                    Cross-region reference check
da42:    testq   %r10, %r10
da45:    je      0x22
da47:    testl   %ebx, %ebx
da49:    je      0x1e                           NULL pointer check
da4b:    shrq    $0x9, %r11
da4f:    movabsq $0x7f48d0c00000, %rdi          Card address calculation
da59:    addq    %r11, %rdi
da5c:    nopl    (%rax)
da60:    cmpb    $0x2, (%rdi)                   Young gen. check
da63:    jne     0x166
  ...
dbcf:    movq    0x48(%r15), %r10               Thread-local dirty card queue
dbd3:    movq    0x58(%r15), %r11
dbd7:    lock                                   Memory barrier
dbd8:    addl    $0x0, -0x40(%rsp)
dbdd:    nop
dbe0:    cmpb    $0x0, (%rdi)                   Dirty card check
dbe3:    je      -0x180
dbe9:    movb    %r12b, (%rdi)                  Dirty the card
dbec:    testq   %r10, %r10
dbef:    jne     0x21                           Enqueue dirty card
dbf1:    movq    %r15, %rsi
dbf4:    nopl    (%rax,%rax)
dbfc:    nop
dc00:    movabsq $0x7f48f236f860, %r10
dc0a:    callq   *%r10
dc0d:    jmp     -0x1a9
dc12:    movq    %rdi, -0x8(%r11,%r10)
dc17:    addq    $-0x8, %r10
dc1b:    movq    %r10, 0x48(%r15)
dc1f:    nop
dc20:    jmp     -0x1bc
```

Figure 3: Post-write barrier x86_64 assembly code produced by C2 compiler

2.3.2 Refinement

Refinement is a mechanism of remembered set maintenance employed by the Garbage-First collector. The task of refinement is examining dirty cards from the card table, identifying cross-region reference locations and updating the remembered sets of target regions. Refinement is normally performed concurrently to the mutator, however self-refinement is also possible. Figure 4 demonstrates a high-level scheme of concurrent refinement operation. The process starts with the post-write barrier dirtying a card (step `(1)` on the figure).

The mechanism relies on *Dirty Card Queue Set (DCQS)* [48] to store pending card addresses, accumulating those in buffers and maintaining a list of buffers ready for refinement. Buffers are populated on per-thread basis by the post-write barrier (step `(2)` on the figure). Once a buffer fills up, the post-write barrier invokes the runtime subroutine, which attaches the buffer to DCQS and allocates a buffer for new writes by the barrier. If number of cards pending refinement rises above certain threshold after the buffer was

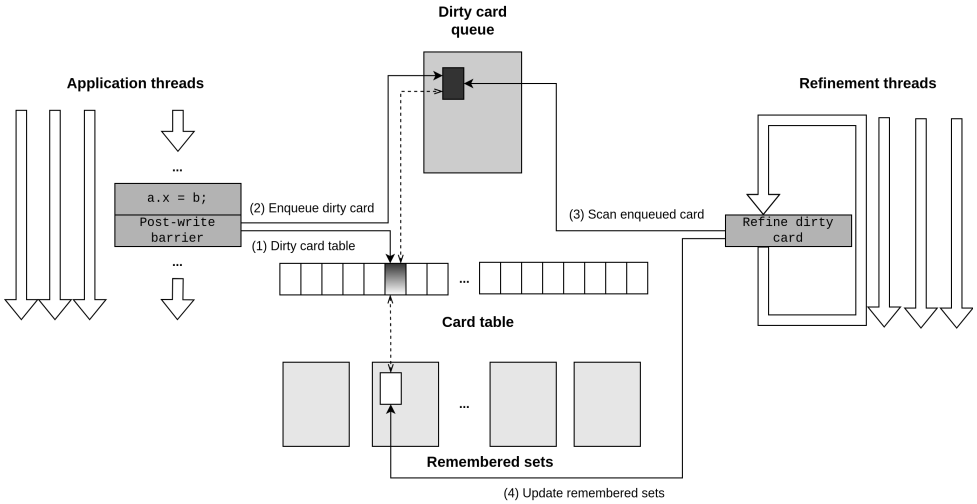


Figure 4: Conceptual scheme of concurrent refinement in G1

attached, the mutator refines the buffer itself (*self-refinement*).

For concurrent refinement G1 maintains a set of parallel threads. The number of active refinement threads is scaled automatically within a pre-configured range. Threads attempt periodic dequeue of card buffers from DCQS. If a buffer is successfully obtained, refinement of the buffer is invoked by the respective thread.

Refinement of a card buffer starts with identifying cards that need refinement and clearing values of these cards. A certain number of cards might not need the refinement; these cards include those that were already refined by other threads, as well as cards contained in a hot card cache — G1 maintains a cache of frequently dirtied cards and avoids refinement of those. Cards can be added to or evicted from hot card cache throughout the refinement process. The rest of the cards in a buffer are refined (step `(3)`) by scanning respective heap ranges and identifying locations of "interesting" references. If a region targeted by such reference is tracked (i.e. it is planned for evacuation), the reference is added to respective remembered set (step `(4)`). Mutator self-refinement follows the same logic, with the difference that refinement is performed in the context of mutator thread.

As a result of the refinement process, dirty cards produced by the mutator are examined and relevant remembered sets are updated. This minimizes the amount of necessary heap root identification work during evacuation.

2.3.3 Evacuation process

The Garbage-First collector relies on stop-the-world pauses to perform live object evacuation and space reclamation. Evacuation is performed by copying live objects from collection set regions to other regions on the heap, with all references to copied object being updated. Evacuating an object might also involve its promotion: based on the age of the object, it may be copied from the young generation region to the old generation. Objects that have not yet reached the age threshold are copied to survivor regions. G1 reclaims the region once all live objects have been evacuated. G1 does not perform evacuation for humongous objects. Due to the design of G1 collector, a stop-the-world pause is necessary to ensure atomicity of changes on the heap from the mutator viewpoint.

The evacuation process in G1 consists of several sequential stages, and involves substantial amount of bookkeeping activities that record the garbage collector behavior.

Recorded statistics are used to both influence heuristic-driven behavior of G1, as well as to provide observability to profiling tools. The evacuation algorithm is the same for both young-only and mixed garbage collections [49], whereas the implementation of full garbage collections is different and is out of scope of this thesis.

Pre-evacuation stage consists of several preparation activities that include flushes of per-thread data structures to produce a consistent global view, calculation of collection set with respect to current pause time goals, preparation of the allocator to initialize allocation regions as evacuation destination. After the pre-evacuation stage finishes, G1 sets up data structures holding information for parallel evacuation threads and proceeds to the initial collection set evacuation.

Evacuation of the initial collection set encompasses several sequential steps:

1. Heap root merge — gathers dirty cards from collection set remembered sets, hot card cache, dirty card queue buffers and merges these cards onto the card table. As a result, all locations potentially storing references to live objects within the collection set are recorded on the card table, thus turning it into a unified remembered set for collection set regions. Furthermore, the merge process implicitly de-duplicates cards found in different sources.

   During heap root merge for subsequent scan purposes, G1 augments the card table with an additional level. The second level of the card table — chunk table — marks chunks, represented as boolean values. Each chunk represents a distinct fixed-size range of cards on the card table. Whenever any card from the range gets dirtied, the respective chunk gets marked. G1 also maintains a dirty region list which includes regions whose respective chunks were marked. Both chunk table and dirty region list are updated during the heap root merge stage. Thus G1 is able to avoid full card table processing later.

   The ability to identify complete set of cards for merging onto the card table is, however, dependent on guarantees provided by the write barrier and refinement mechanisms of G1. The post-write barrier ensures that all relevant modifications of the heap get marked on the card table, and dirty cards are added to the queues. The refinement mechanism ensures that dirty cards produced by the post-write barrier get scanned and added either to remembered sets, or to the hot card cache. Therefore, changes to these mechanisms have direct effect on the heap root merge, which is particularly important in the context of this thesis.

2. Heap root scan — identifies references pointing to reachable objects within collection set regions. The process starts with traversal of virtual machine roots and evacuation of directly referenced objects. Then heap root scan iterates the card table populated during heap root merge phase, scans dirty cards and identifies actual locations of cross-region references. Locations of references that point into the collection set regions are saved into task queues, whereas remaining cross-region references get preserved for future collections using the redirtying mechanism. Finally, code roots contained in collection set region structure are also traversed. As a result of this phase, the complete set of references to live objects within the collection set regions is obtained.

3. Evacuation — iterates task queues built during the root scan phase and performs the

copying of live objects to other regions. Evacuated objects are scanned to identify other live objects within collection set regions, which are also subsequently evacuated.

Evacuation of the optional collection set follows the evacuation of the initial collection set and is based on the same algorithm with several adjustments. This stage is performed only in case the pre-evacuation phase determined optional collection set regions and there is pause time budget left until the budge is exhausted. Notable evacuation algorithm adjustments are following:

- Heap root merge only takes into account remembered sets of optional collection set regions. Dirty cards from card buffers, hot card cache, humongous region candidates have already been processed as part of initial collection set evacuation.
- Heap root scan — virtual machine roots have already been processed during the initial collection set evacuation, so for optional collection set only merged dirty cards scan and collection set regions processing happens.

Post-evacuation tasks include numerous activities necessary to finalize changes done during the evacuation pause, reclaim memory and process gathered statistics. In the context of this thesis, following post-evacuation tasks are particularly relevant:

- Card table cleanup — the card table is being cleaned from dirty cards which were processed throughout the evacuation. The cleanup is based on the dirty region list produced during heap root merge phase.
- Redirtying — locations of cross-region references that were found or newly introduced during evacuation need to get preserved in order to ensure correctness of subsequent incremental evacuations. During evacuation these references are saved within a *Redirty Cards Queue Set (RDCQS)*. During the post-evacuation stage cards in the RDCQS are dirtied on the card table, and buffers of cards from that data structure are merged into the refinement dirty card queue. These cards can, therefore, be processed by refinement mechanism during mutator time or during heap root merge stage of the next garbage collection.

2.4 Problem analysis

Detailed description of background information provided in this chapter enables a more precise and complete reformulation of problem. The throughput of the G1 collector depends on multitude of factors, including the design of the write barrier and guarantees it provides to other parts of the garbage collector. This thesis focuses on improving the throughput by exploring alternative post-write barriers and their implications on G1.

The post-write barrier, described in section 2.3.1, is conceptually complex. This complexity is necessary to fulfill assumptions made during evacuation and refinement (card marking & dirty card enqueue), filter out non-“interesting” references (conditional checks), and interact with concurrent refinement threads (memory barrier instruction), therefore it cannot be optimized out in general. Only in specific cases, when the compiler can deduce certain properties of reference store operation, the barrier can be eliminated, partially or completely. For instance, the C2 compiler provides special handling for the `Object.clone()` method with reduced number of conditionals and without a memory

barrier. Furthermore, the barrier can also be eliminated if the respective allocation also can.

Prior research on write barriers has demonstrated that barriers equivalent to the one used by G1, may cost as much as 12% in terms of throughput [76]. At the same time, simpler write barrier variants reduce that number to $2-6\%$ [5, 74]. Furthermore, decisions on separation of fast- and slow-paths of the barrier are known to impact both runtime and compilation performance [4]. Figure 3 demonstrates decomposition of the x86_64 assembly code produced by the optimizing C2 compiler. The code fragment consists of 33 assembly instructions, which are split into two fragments — inline fast-path conditional checks and out-of-line card marking and dirty card enqueue. With exception to previously mentioned cases when the barrier can be eliminated, the post-write barrier portion has to be included along with every reference store operation. While the optimizing C2 compiler [47] is capable of more informed, profile-based decisions in code generation to optimize performance, the barrier code produced by C1 [46], template interpreter and code stubs [45] is mapped to machine instructions in a straightforward manner. The impact of complex post-write barrier can manifest directly in extra throughput costs for mutator, as well as create CPU instruction cache pressure and overhead for JIT compiler. Precise costs of the post-write barrier are dependent on heap activity profile of the running application. Applications with higher proportion of reference writes that modify old generation objects and cross regions can be expected to execute slow-path of the post-write barrier frequently.

Within current G1 collector design, the costs associated with the complex barrier are justified by guarantees it establishes for other parts of the collector. Incremental heap size-agnostic evacuation is enabled by concurrent refinement which is, in turn, reliant on write barrier enqueueing dirty cards into DCQS. Concurrent refinement itself relies on the write barrier that includes memory barrier instruction. Filtering out same region references, null references and references to young generation reduce load on the refinement mechanism. The original design of G1 [14] states that trading certain amount of throughput for predictable pause times is deemed acceptable.

Therefore, changes introduced into the post-write barrier to achieve higher mutator throughput create important implications for the rest of G1. Reduction of costly post-write barrier steps would incur a shift of respective work into the evacuation pause. Moreover, changes to the card enqueueing procedure might draw concurrent refinement obsolete, whereas dropping the young generation card check would remove the need to maintain respective marking on the card table. Thus, throughput improvements achieved by updates to the post-write barrier need to be, at minimum, accompanied by changes to refinement mechanism and heap root merge phase of evacuation, potentially shifting the balance between throughput and pause time goals. Both throughput and pause time need to get evaluated in order to get more objective conclusions on alternative post-write barrier effects.

Another aim of this thesis is the implementation of a dynamically switched barrier variant. In addition to aforementioned challenges, barrier switch at runtime introduces extra ramifications for both G1 collector and HotSpot virtual machine. G1 shall be able to adjust collector behavior based on current barrier, maintain consistency of its data structures upon switch, collect and process needed statistics in order to use relevant heuristics for the switch. HotSpot shall offer mechanisms to switch barriers in JIT-compiled code and template interpreter consistently and atomically. As the extent to which barrier switch is possible within JDK 20 code base is unknown prior to actual implementation attempt, obtained results will lead to additional conclusions of HotSpot versatility in that respect.

Chapter 3

Implementation

This chapter describes the design and implementation of throughput-oriented changes in the Garbage-First collector. The chapter begins with analysis and derivation of throughput-oriented post-write barrier variants, then describes necessary adaptations of G1 collector. Afterwards, viable implementation variants, combining throughput-oriented barriers with the collector adaptations, are presented. Finally, a detailed description of barrier dynamic switching challenges, implementation and limitations is provided.

All changes to the OpenJDK project described in this chapter are available at [59].

3.1 Throughput-oriented barriers

The main goal of this thesis is the exploration of alternative throughput-oriented write barriers for G1. The primary approach is optimization of the current post-write barrier used by the G1 collector. The description provided in section 2.3.1 reveals significant complexity. That complexity is innate in the current G1 collector design, thus mechanical optimization is not possible — in fact, certain techniques are already used to reduce the number of instructions in machine code. Thus as part of this throughput-focused optimization the steps executed in the post-write barrier have to be reconsidered.

Proposals for a throughput-oriented redesign of the post-write barrier have already been made within the OpenJDK development community [7, 8]. The core idea of the discussion is stripping the most computationally-expensive steps from the post-write barrier completely, producing a simpler barrier with less special cases and in-place processing. The general approach taken in the scope of this thesis is the same — substantial minimization of the post-write barrier. Alternative visions that concentrate on optimizing specific post-write barrier parts, such as [68], preserve the overall post-write barrier algorithm and structure. The background (2.4) section of this thesis shows that shorter minimalist barriers have lower throughput impact than the one used by G1 at the moment. Therefore, a substantial redesign of the post-write barrier has been selected over alternative, limited optimization approaches.

Examples of barrier implementations provided in this chapter are targeted towards the x86_64 instruction set architecture. The implementation effort is also primarily focused on that architecture. Nevertheless, proposed changes shall be portable to other CPU architectures supported by HotSpot as well. Throughout this chapter the normal G1 post-write barrier used as of JDK 20 is referred to as a *"baseline"*.

3.1.1 Post-write barrier decomposition

The post-write barrier algorithm steps listed in the section 2.3.1 can be grouped into parts as following:

- Pre-filtering conditions — a set of pre-conditions for the references that filter out those that are not "interesting" for further processing. G1 includes 4 such conditions, which are placed at the beginning of the post-write barrier. These conditions both reduce the amount of work performed during refinement and evacuation, as well as avoid frequent executions of the barrier slow path.
  At the same time, the conditions also have potential to harm throughput: in modern processors branch mispredictions are known to incur significant performance penalty [16]. Besides that, machine code for the conditions also creates additional instruction cache pressure. Code generated for x86_64 architecture depicted on the figure 3 shows 8 instructions (22 bytes) related to the pre-filtering conditions placed onto the fast-path. The possible impact of pre-filtering conditions on the post-write barrier throughput is not immediately obvious, and so is their benefit during evacuation. Removal of these conditions has a potential of improving the post-write barrier throughput without compromising the evacuation correctness and completeness.

- Card marking — the core of the post-write barrier that calculates and dirties a card that corresponds to reference store location. Within the current G1 design, the card table is an essential data structure, acting as an intermediate store for dirty cards during mutator time and as a remembered set of the collection set regions during evacuation, therefore card marking is a fundamental part of the post-write barrier. The card marking approach is known to have moderate throughput costs [5, 74]. Furthermore, the card marking algorithm conceptually provides constant algorithmic complexity regardless of reference store frequency. At the same time, on multiprocessor systems unconditional card marking is known to introduce cache contention [15]. For x86_64, unconditional card marking consists of 4 machine instructions (20 bytes).

- Refinement — parts of the barrier related to collaboration with concurrent refinement threads and dirty card queue maintenance. This group includes memory barrier instructions, management of the sequential store buffer and the call into the G1 runtime. Potential throughput problems include non-deterministic complexity of the dirty card enqueue operation, performance penalty due to the store-load barrier, pressure on the instruction cache and additional JIT compilation time complexity. For x86_64, this group consists of 17 instructions (69 bytes), and is by far the largest part of the post-write barrier. Partial elimination of these operations is problematic, because the group itself is tightly cohesive and will not be able to serve its purpose partially.
  For instance, removal of dirty card enqueue operations in isolation will draw the memory barrier useless, whereas isolated removal of the memory barrier is not possible due to concurrent and self-refinement; complete disablement of refinement is possible, however, by itself it will have minuscule impact on the post-write barrier and degrade heap root merge pause times. This part of the barrier, therefore, is the main candidate for complete removal.

Based on the analysis above, all throughput-oriented post-write barrier variants need to include card marking as an essential part, might include certain pre-filtering conditions,

and refinement-related operations will be omitted in all barrier variants.

3.1.2 "Raw Parallel" barrier

The purpose of the first post-write barrier approach is being as minimal as possible within the barrier decomposition analysis provided above, removing everything that is not strictly necessary. The implementation uses the codename "Raw Parallel" for this barrier variant, because it closely resembles the barrier used by the Parallel Collector [43].

The only mandatory element of the post-write barrier is card marking operation, which is unconditional and consists of 3 steps: card index calculation via bit shift, card address calculation by offsetting the index to the beginning of the card table, and dirtying the card. This variant omits other parts of the current post-write barrier — pre-filtering conditions and refinement — completely. The resulting barrier is much shorter and can be inlined by JIT compiler in entirety.

While this simplification of the post-write barrier is promising in terms of throughput gains, basic unconditional card marking barrier also presents some additional challenges and requires extra attention towards correctness. Besides generic adaptations to G1 collector described in the section 3.2, following aspects are important:

- As a mitigation for cache contention on multiprocessor systems (section 2.2.3) a conditional check of card dirtiness is reintroduced back into the barrier for such systems. Presence of the reintroduced check is enabled using a command line flag of the JVM. Thus, the end-user is able to make the card marking conditional is if it is deemed beneficial for target architecture. This approach matches behavior of the Parallel Collector that uses `UseCondCardMark` for the same purpose.

- The proposed version of the barrier lacks any memory synchronization primitives, so there is no specific globally visible order of heap and card table modifications by the mutator. This approach is correct as long as there is no refinement happening concurrently to the mutator (memory barrier instruction description in the section 2.3.1). For the "Raw Parallel" barrier to work, it is essential to either disable the refinement completely, or adjust it so that it does not clean scanned cards. An alternative is reintroducing the memory barrier instruction back into the post-write barrier, which is unfeasible due to throughput goals of the changes.

Figure 5 demonstrates a possible implementation of the "Raw Parallel" barrier for the x86_64 architecture and compares it to the baseline G1 post-write barrier. Four instructions that belong to unconditional card marking barrier are highlighted with darker background, whereas two instructions that implement dirty card check condition are highlighted in light gray — these are included only if `+UseCondCardMark` option is passed to JVM.

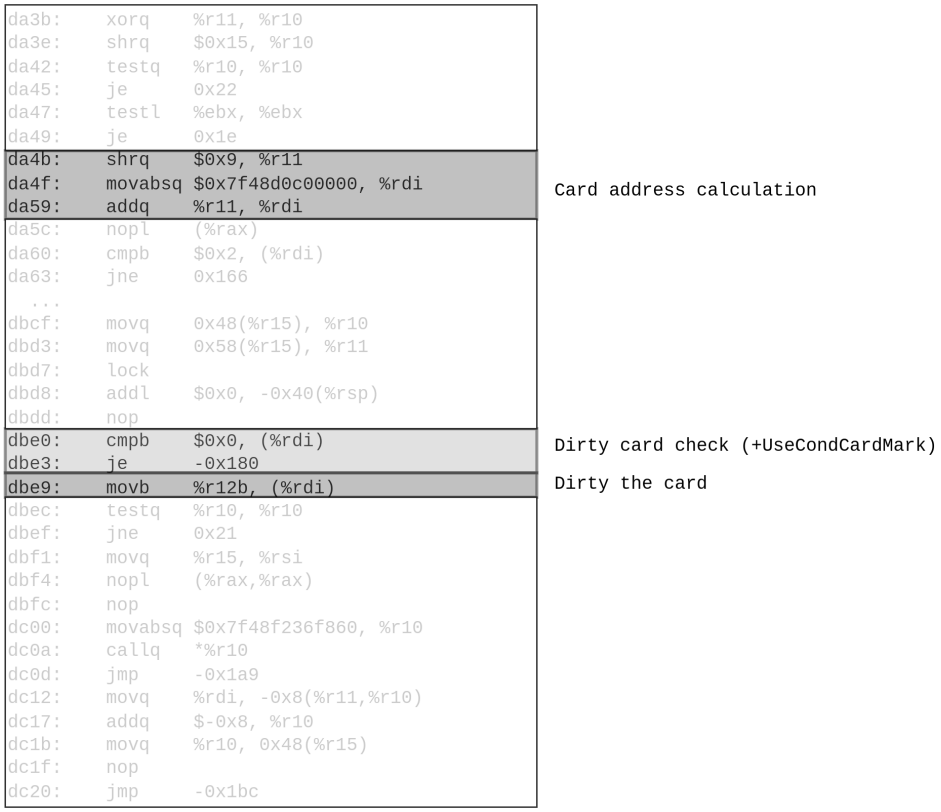


Figure 5: "Raw Parallel" barrier compared to the baseline for x86_64 architecture

3.1.3 "Long" barrier

The second variant of the throughput-oriented barrier builds upon the "Raw Parallel" barrier, augmenting it with pre-filtering conditions. As noted in section 3.1.1, the impact of pre-filters on barrier throughput is unknown, therefore investigation of such barrier variant is valuable. This barrier variant has the codename "Long" as it extends over the minimal "Raw Parallel" barrier. Pre-filtering conditions need to be examined separately with respect to their usefulness in reducing amount of work done at heap root merge and scan phases of the evacuation.

- The cross-region reference check is valuable because only a fraction of references produced by the mutator cross region boundaries and only those need to be considered by the collector. Placing the cross-region reference check at the beginning of the barrier eliminates the need to process such references further.
- Null pointer check is also preserved due to the same reasons.
- The young generation check has been originally introduced [68] to avoid executing the expensive memory barrier instruction all the time. With the memory barrier removed entirely from the throughput-oriented write barriers, this condition can be omitted.
- Dirty card check is useful as it reduces cache contention by filtering out modifications to the same card, therefore it is included in this barrier.

Based on the analysis above, the "Long" barrier variant consists of 3 pre-filtering conditions and the card marking operation, but still excludes the memory barrier and

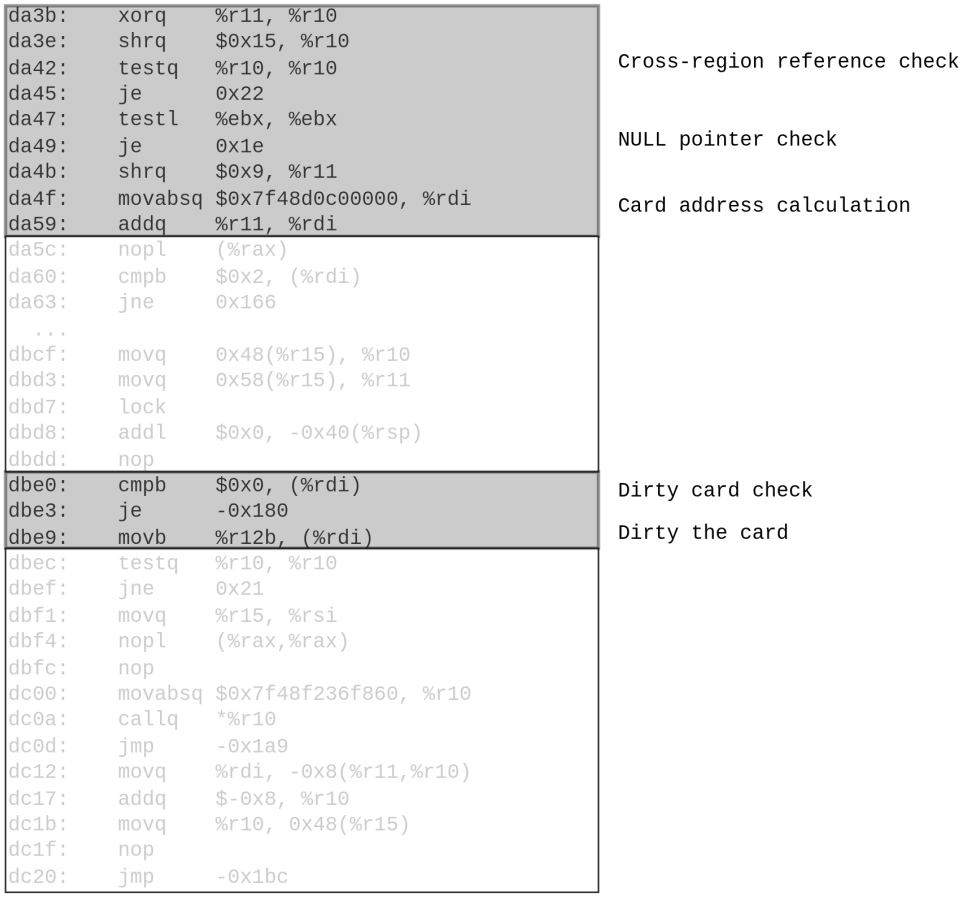


Figure 6: "Long" barrier compared to the baseline for x86_64 architecture

refinement. Due to the missing memory barrier, this variant is also prone to memory operation ordering issues described in the section 3.1.2. Figure 6 shows a possible implementation of "Long" barrier for the x86_64 architecture, comparing it to the baseline G1 post-write barrier. Despite including extra checks, the "Long" barrier is nonetheless 2.75 times shorter than the baseline barrier, consisting of 12 machine instructions (45 bytes).

3.1.4 Post-write barrier implementation

This section provides details on the implementation of throughput barrier variants. The implementation needs to be done for the template interpreter, C1 and C2 compilers separately. Due to the fact that the newly derived barrier variants are strict subsets of the baseline G1 barrier, conditional disablement of respective operations within the barrier code is sufficient for the most part. Nevertheless, several important details of the implementation process need to be mentioned.

The general approach to the implementation relies on using conditional compilation via C++ preprocessor to disable post-write barrier parts that do not belong to either of throughput barrier variants. A set of C++ macros is defined to distinguish between configurations. Barrier parts that vary between variants (pre-filtering conditions) are controlled using normal C++ `if` statements. In order to avoid overhead of newly introduced conditional branches, conditions rely on HotSpot `develop` flags — command line flags that can be configured at runtime only for debug builds, but in release (optimized)

builds have pre-configured constant values. Therefore, newly introduced branches can be optimized out by C++ compiler based on compile-time configuration and do not bring any overhead. As a consequence, no runtime configuration is possible and each barrier variant requires own JVM build, yet all variants including baseline can co-exist within the same code base.

The interpreter implementation [45] of the G1 write barrier relies on the `MacroAssembler` class, whose methods, in general, correspond to target CPU instruction set architecture opcodes. It uses simple linear code flow, which is controlled by jump instructions. Thus, the implementation of barrier changes for the interpreter is trivial — the assembly code of barrier variants is analyzed and deconstructed in a manner similar to the one shown on figures 5 and 6, and unneeded operations are removed in accordance with general approach described above. Listing 2 shows an example of disabling post-write barrier part that is not present in any of the throughput barrier variants. Listing 3 shows a part of barrier (pre-filtering conditions) that is omitted for "Raw Parallel" variant, but included otherwise. Array reference write stubs are updated in identical manner.

```
#ifdef DISABLE_TP_REMSET_INVESTIGATION
  __ cmpb(Address(card_addr, 0), G1CardTable::g1_young_card_val());
  __ jcc(Assembler::equal, done);

  __ membar(Assembler::Membar_mask_bits(Assembler::StoreLoad));
#endif
```

Listing 2: Young generation check operation disabled for throughput barriers (interpreter)

```
TP_REMSET_INVESTIGATION_ONLY(if (!G1TpRemsetInvestigationRawParallelBarrier)) {
    // Does store cross heap regions?

    __ movptr(tmp, store_addr);
    __ xorptr(tmp, new_val);
    __ shrptr(tmp, HeapRegion::LogOfHRGrainBytes);
    __ jcc(Assembler::equal, done);

    // crosses regions, storing NULL?

    __ cmpptr(new_val, NULL_WORD);
    __ jcc(Assembler::equal, done);
}
```

Listing 3: Pre-filtering conditions disabled for "Raw Parallel" barrier variant (interpreter)

The C1 compiler implementation of the G1 write barrier uses an approach similar to the interpreter. C1 code splits the post-write barrier in two parts: fast- and slow-path, where slow-path is located within the same code unit as interpreter code and uses a mechanism derived from `MacroAssembler` for code generation. The fast-path [46] is implemented separately using `LIRGenerator`, which offers a higher level intermediate representation (IR) of code. Despite being on a different abstraction level, control flow implementation for the post-write barrier fast-path is still based on jump and branch primitives, thus the throughput-oriented barrier changes follow the same logic as with the interpreter. Listing 4 shows disabled pre-filtering conditions for "Raw Parallel" barrier variant.

```
TP_REMSET_INVESTIGATION_ONLY(if (!G1TpRemsetInvestigationRawParallelBarrier)) {
    LIR_Opr xor_res = gen->new_pointer_register();
    LIR_Opr xor_shift_res = gen->new_pointer_register();
    if (TwoOperandLIRForm) {
        __ move(addr, xor_res);
        __ logical_xor(xor_res, new_val, xor_res);
        __ move(xor_res, xor_shift_res);
        __ unsigned_shift_right(/* ... */);
    } else {
        /* ... */
    }

    __ cmp(lir_cond_notEqual, xor_shift_res, LIR_OprFact::intptrConst(NULL_WORD));
}
```

Listing 4: Pre-filtering conditions disabled for "Raw Parallel" barrier variant (C1 compiler)

The C2 compiler utilizes a sophisticated sea-of-nodes intermediate representation, which separates data and control flows and does not define explicit scheduling of instructions. This makes omitting barrier parts easier, because only nodes that explicitly belong to the control flow graph need to be disabled — the optimizer is capable of eliminating unused data flow elements automatically. Listing 5 shows how cross-region reference pre-filtering condition is eliminated for the "Raw Parallel" barrier variant: only the `if` condition node is explicitly disabled, whereas its parameter nodes stay unchanged (compare to the listing 3 which exhibits linear code flow).

```
Node* cast =  __ CastPX(__ ctrl(), adr);
Node* xor_res =  __ URShiftX ( __ XorX( cast,  __ CastPX(__ ctrl(), val)), __
↪  ConI(HeapRegion::LogOfHRGrainBytes));

TP_REMSET_INVESTIGATION_ONLY(if (!G1TpRemsetInvestigationRawParallelBarrier)) __
↪  if_then(xor_res, BoolTest::ne, zeroX, likely);
    /* ... */
TP_REMSET_INVESTIGATION_ONLY(if (!G1TpRemsetInvestigationRawParallelBarrier)) __
↪  end_if();
```

Listing 5: Pre-filtering condition disabled for "Raw Parallel" barrier variant (C2 compiler)

Nonetheless, the C2 compiler introduces different challenge for the throughput barrier implementation. The write barrier implementation for the C2 compiler has to provide a method for barrier elimination in the IR graph. The method used for baseline G1 barrier elimination exploits the fact that the baseline post-write barrier contains a top-level condition which wraps the rest of the barrier. The elimination routine simply replaces the condition input with constant *false* and relies on the optimizer removing barrier nodes because they would never execute. This approach works for the "Long" barrier as well, however it breaks for the unconditional "Raw Parallel" barrier variant. Instead, the implementation for the "Raw Parallel" traverses barrier nodes and the operation that produces the side-effect — card marking — is replaced. This enables the optimizer to eliminate the rest of the barrier as it has no externally visible effects. Listing 6 demonstrates the gist of this approach.

```
if (G1TpRemsetInvestigationRawParallelBarrier) {
    Node *shift = node->unique_out();
    Node *addp = shift->unique_out();
    for (DUIterator_Last jmin, j = addp->last_outs(jmin); j >= jmin; --j) {
        Node *mem = addp->last_out(j);
        macro->replace_node(mem, mem->in(MemNode::Memory));
    }
    /* ... */
}
```

Listing 6: Barrier elimination for "Raw Parallel" barrier variant (C2 compiler)

Barrier runtime changes are not strictly required for throughput-oriented barrier variants, because no new runtime calls are introduced by the barrier. However, due to the fact that throughput barriers omit calls of dirty card enqueue, corresponding runtime routines are augmented with asserts to ensure that the routines are never invoked. HotSpot defines the `ShouldNotCallThis` macro for this purpose.

Resulting code generated by C2 compiler for "Raw Parallel" and "Long" barriers is provided in appendices A.2 and A.3 respectively. In both cases, the write barrier is short enough to be generated as a linear sequence of machine code (i.e. without splitting it into fast- and slow-path fragments) by the optimizing compiler. Furthermore, for "Raw Parallel" barrier the C2 compiler is often able to generate shorter card marking code by using an extra register as an indirection base — this is a consequence of lower register pressure induced by the minimal barrier.

3.2 Garbage-First collector adaptation

Use of throughput-oriented barriers proposed in section 3.1 necessitates updates in the G1 collector to make it function correctly. The G1 collector heap root scan phase and dirty card refinement (section 2.3.3) are affected by the throughput-oriented barriers. This section provides a detailed description and rationale of the changes.

The general approach to implementation of collector changes is the same as with barriers. The same set of C++ macros is used to distinguish between the baseline G1 behavior and throughput adaptations. All changes co-exist within the same code base with no additional runtime overhead, but every configuration is static and requires a separate JVM build.

3.2.1 Heap root scan

The heap root merge phase of the baseline G1 evacuation process relies on the assumption that a complete set of root locations is contained within the remembered sets, dirty card queues and hot card cache. Cards contained within these data structures are dirtied on the card table and respective chunks are marked on the chunk table. The heap root scan phase relies on chunk table markings to selectively scan the card table (section 2.3.3). The baseline G1 post-write barrier is responsible for updating the dirty card queues and cooperating with the concurrent refinement to maintain the remembered sets (section 2.3.1).

The "Long" and "Raw Parallel" barriers omit any interaction between the mutator and remembered set maintenance mechanisms. The throughput-oriented barrier variants still keep root locations on the card table, however the remembered set, dirty card queues and chunk table are not updated anymore. Therefore, G1 needs to scan all cards of non-collection set regions to determine a complete set of root locations. The implementation reuses existing card table scan mechanism by dirtying the whole chunk table in advance, thus enforcing a full card table scan.

Listing 7 shows code that is necessary to ensure card table scan for a specific region. G1 implements work parallelization, and code from listing 7 is being a part of worker task closure, which is executed for different regions in parallel as part of heap root merge phase of evacuation.

```
1 if (hr->in_collection_set() || !hr->is_old_or_humongous_or_archive()) {
2     return;
3 }
4
5 _scan_state->add_dirty_region(hr->hrm_index());
6 size_t const first_card_idx = _ct->index_for(hr->bottom());
7 _scan_state->set_chunk_range_dirty(first_card_idx, HeapRegion::CardsPerRegion);
```

Listing 7: Chunk table and dirty region list modification for a heap region

Implemented full card table scan approach has an advantage of conceptual simplicity: instead of re-implementing the heap root scan phase, the implementation expands existing mechanisms and requires minimal changes to the code base. Furthermore, it also fixes card table clearing — it also uses dirty region list to identify card table fragments to clear. At the same time, full card table scan has an important consequence of shifting G1 throughput and pause time trade-off by increasing the duration of heap root scan phase.

3.2.2 Refinement

The concurrent refinement mechanism of G1 collector requires considerable adaptation in order to work with throughput barriers. The new barrier variants do not cooperate with concurrent refinement threads and do not perform self-refinement, so the majority of cards — the ones dirtied by the mutator — cannot be refined. Although refinement is still partially possible for dirty cards produced during evacuation, its logic needs to be reworked in order to avoid conflicts with the barrier. Potential solutions of that issue include scope limiting of adapted concurrent refinement, doing refinement during garbage collection pause or disabling the refinement completely.

Concurrent refinement interaction with throughput barriers creates store operation reordering problem which can result in missing card table markings. In the baseline G1 post-write barrier, the reordering issue has been solved by using a memory barrier instruction (section 2.3.1). Both throughput write barrier variants lack the memory barrier and are prone to the issue, however specific details of memory ordering problems differ:

- For "Long" barrier variant, the issue appears when card value load for dirtiness check is reordered prior to the reference store operation. In such case the mutator might load the value while the card is still unrefined and dirty, while reference write effect appears after the refinement happened. The barrier will not dirty the card then, thus the card table marking will be missing.

- For "Raw Parallel" barrier variant, if the effect of card dirtying appears prior to reference write, the concurrent refinement thread might clear and scan the card without encountering newly updated reference, also leading to the same problem[1].

Therefore, concurrent refinement needs to be significantly adapted in order to get used along with throughput barriers. There are several adaptation approaches:

- *Partial concurrent refinement* can be done via introduction of a special "refine concurrently" card marking for cards redirtied after evacuation. The dirty card queue is populated as part of post-evacuation process with the "refine concurrently" cards and processed by concurrent refinement threads. Concurrent refinement only needs to consider cards with that marking, while actually dirty cards produced by the mutator are ignored by the concurrent refinement and will be scanned during the collection pause. Furthermore, refinement does not clear cards prior to scanning, so it cannot conflict with the barrier. Cards that were not refined before the next garbage collection get dirtied at heap root merge phase based on dirty card queue contents. This adaptation ensures that the concurrent refinement never conflicts with the barrier: even in case of store reordering the card will never get cleared during mutator time, and cards that were modified by the mutator are not refined concurrently. Moreover, the special marking for redirtied cards ensure that there is no double processing during heap root scan for cards that were refined — such cards are not considered dirty and are not scanned, unless they appear in collection set region remembered sets and were not dirtied by the mutator.

- *Post-evacuation refinement* refines cards produced during the evacuation phase as post-evacuation task. The main disadvantage of this approach is degradation of pause times with no apparent benefit for throughput. The number of cards pending for refinement cannot be easily estimated, so it also harms predictability of the garbage collection pause. Furthermore, cards refined after evacuation and dirtied during subsequent mutator time get scanned once again in the root scan phase of the next garbage collection, in addition to refinement in this phase.

- *Disabling refinement* implies that all dirty cards regardless of their origin stay on the card table until the next garbage collection, when they get scanned. The role of remembered sets in heap root merge phase is reduced, as they only include information on heap roots identified during remembered set rebuild after concurrent marking finishes with no later updates. Implementation of this approach is trivial within current G1 architecture: the number of concurrent refinement threads is set to zero, and the redirtying post-evacuation task is updated to skip adding cards to dirty card queues.

  This approach further shifts the balance between throughput and collection pause time: refinement does not use any processing resources, yet more cards need to get scanned during the root scan phase. Compared to post-evacuation refinement, disabled refinement is not prone to double-scanning cards — card table naturally de-duplicates writes to the same card, so no card is scanned more than once between two evacuations.

[1]Some architectures including Intel x86_64 ensure that store operations are seen in program order [24], so the issue may not appear in practice.

3.2.3 Other adaptations

While full card table scan and refinement changes described in the previous sections are strictly necessary in order to use throughput barriers with G1, there are also several optional minor modifications which are enabled by the new post-write barriers.

Special young generation marking on the card table can be omitted, as neither of barrier variants includes the corresponding check. In fact, cards that correspond to young generation regions need no marking at all, because the young generation is always a part of collection set regardless of garbage collection type. However, card dirtiness checks are included into both the "Long" barrier algorithm and some of G1 collector routines, thus marking all young generation cards as dirty would lead to lower number of card table modifications.

Memory barriers that ensure synchronization between the concurrent refinement and the rest of G1 collector are present not only within the baseline post-write barrier, but also in several other places in the garbage collector. With the refinement adaptations described in the section 3.2.2, these barriers can be removed as well because modified refinement variants require no particular memory operation ordering.

Chunk table modification as part of the post-write barrier removes the requirement of doing full card table scan. Instead of dirtying the whole chunk table during heap root merge phase as described in the section 3.2.1, dirty chunks can be marked by the mutator in the same manner as dirty cards. Therefore, during the root scan phase of evacuation only those parts of card table that actually contain dirty cards will get scanned. The primary advantage of this adaptation is reduction of root scan phase time, thus mitigating collection pause time impact of throughput barrier variants. However, adding chunk table marking as part of the post-write barrier is expected to harm mutator throughput, and due to the fact that chunk table is very dense representation of the whole heap, can lead to high amount of false sharing as described in [15]. Listing 8 shows an implementation of chunk dirtying for the interpreter barrier code. The implementation uses the same approach as with card dirtying, but has different bit shift value and base address, corresponding to the chunk table.

```
1  if (G1TpRemsetInvestigationDirtyChunkAtBarrier) {
2      G1RemSet* rem_set = G1BarrierSet::rem_set();
3      uint8_t const chunk_shift = rem_set->region_scan_chunk_table_shift();
4      intptr_t const chunk_table_base = rem_set->region_scan_chunk_table_base();
5
6      __ pop(card_addr);
7      __ shrptr(card_addr, chunk_shift);
8      __ movptr(cardtable, chunk_table_base);
9      __ addptr(card_addr, cardtable);
10     __ movb(Address(card_addr, 0), true);
11     __ jmp(done);
12 }
```

Listing 8: Chunk dirtying implementation for the post-write barrier (interpreter)

Use of chunk dirtying has negatively impacted the overall throughput, thus it is not considered further. Chapter 4 includes a brief analysis of the chunk dirtying impact on mutator throughput.

Direct remembered set update avoids redirtying and refining cards produced during the evacuation phase by immediately putting "interesting" references into the corresponding remembered sets. This approach allows a limited degree of remembered set maintenance, while greatly reducing the need for refinement at the same time. Listing 9 shows a simplified version of code that implements this optimization as part of "interesting" reference identification during the evacuation.

```
1 if (G1TpRemsetInvestigationDirectUpdate && !region_attr.is_optional()) {
2     HeapRegion* const hr = _g1h->heap_region_containing(o);
3     hr->rem_set()->add_reference(p, _worker_id);
4 }
```

Listing 9: Direct update of remembered set for non-optional collection set regions

Despite the appeal of this optimization, it is not dependent on the throughput barrier and also applicable to the baseline G1. This thesis focuses solely on throughput optimizations that stem from the alternative write barriers, therefore, this optimization is out of scope.

3.3 Selected implementation variants

Sections 3.1 and 3.2 describe different variants of throughput barriers and G1 collector adaptations, opening up a number of possible throughput-oriented garbage collector configurations.

A throughput-oriented collector configuration can be freely combined from the following aspects:

- Post-write barrier variants — "Raw Parallel" and "Long" throughput barriers have been designed in the scope of this thesis. Both barrier variants require the same changes of G1 collector, and thus are functionally equivalent and interchangeable, but provide different trade-offs with respect to mutator throughput.
- Refinement adaptations — partial concurrent and disabled refinement are viable options, both having different impact on throughput-pause time balance, yet being functionally equivalent and interchangeable. Disabled refinement frees up processing resources dedicated towards refinement during mutator time at the expense of doing scanning at pause time, whereas adapted concurrent refinement partially preserves existing G1 trade-offs. Thus, partial concurrent refinement and disabled refinement are included into different garbage collector variants. At the same time, post-evacuation refinement is prone to double scanning cards at pause time with no apparent benefit in throughput, bringing no conceptual improvement over alternative approaches — this variant is considered unfeasible.
- Chunk dirtying is applicable to any combination of barriers and refinement adaptations, doubling the number of possible configurations. It is expected to affect both throughput and pause times.

Thus, there are $2^3 = 8$ possible configurations of throughput-oriented G1 collector. In order to perform comprehensible benchmarking and analysis of the change impact, only a subset of these configurations has to be chosen. Selection of configurations is based on combined degree of divergence from the baseline G1 behavior for each configuration:

- "Long" variant — G1 configuration that combines "Long" barrier with partial concurrent refinement. This configuration only moderately diverges from the baseline G1, dropping the refinement of cards dirtied by the mutator.
- "Raw Parallel" variant — G1 configuration that combines "Raw Parallel" barrier with disabled refinement, diverging from the baseline more substantially than the "Long" variant. This variant not only drops the refinement completely, but also utilizes minimal card marking post-write barrier, resembling the Parallel Collector in that respect.

Chunk dirtying is not directly related to throughput improvements, so it is not included into either configuration variant. However, a combination of "Raw Parallel" variant and chunk dirtying is used for a limited number of benchmarks to characterize the impact of chunk dirtying in isolation.

It shall be reiterated that changes in collector variants are focused solely on accommodating new barriers, and in all other respects the behavior is still the same as for the baseline G1. Table 1 summarizes "Long" and "Raw Parallel" variants and compares them to the original G1. Appendix B provides details on configuration of OpenJDK builds with throughput-oriented garbage collector.

Features	Original G1	"Long"	"Raw Parallel"
Card marking in the barrier	✓	✓	✓
Pre-filtering in the barrier	✓	✓	
Refinement support in the barrier	✓		
Chunk dirtying in the barrier			
Self-refinement	✓		
Concurrent refinement	✓	Partial[1]	
Post-evacuation refinement			
Full card table scan		✓	✓

[1] Partial concurrent refinement as described in the section 3.2.2

Table 1: Garbage collector variant comparison

3.4 Dynamically-switched barriers

Throughput-oriented G1 collector variants designed in the previous sections have a significant limitation — their configurations are static and cannot be switched either at runtime, or at startup, instead requiring a rebuild of OpenJDK with different compilation flags. Such approach is beneficial for throughput barrier evaluation, because it enables co-existence of different throughput-oriented barriers and adaptations within the same code base, isolating their runtime overhead at the same time. However, this scheme requires having separate builds of OpenJDK, which is impractical, especially if none of the G1 variant is deemed particularly suitable for a specific workload. In such cases an ability to dynamically switch between the baseline and throughput-oriented G1 might be beneficial. This section provides a detailed overview of a prototypical implementation that can dynamically switch barriers, and describes associated challenges and limitations of current HotSpot architecture.

The implementation of dynamic barrier switch involves combining different post-write barriers and their respective garbage collector adaptations in such a way that specific con-

figuration can be selected at runtime. Even though dynamic barrier switch can potentially involve more than two barriers, the discussed implementation focuses on switching between only two variants — the baseline and "Raw Parallel" post-write barriers. From an implementation perspective, switching between multiple types would be conceptually the same, however it incurs additional runtime complexity and overhead in the form of extra conditional branches incorporating barrier-specific garbage collector adaptations. The "Raw Parallel" variant has been chosen for this experiment because it is most different from the baseline G1 in terms of anticipated throughput. With dynamic barrier switching available, G1 effectively gains two modes of operation: baseline and throughput-oriented.

There are two parts to the implementation of dynamic barrier switching: first, refactoring the existing code to allow two barrier implementations in the same virtual machine build, and second, implementing the actual runtime barrier switch mechanism. The refactoring part is, in fact, trivial and consists of converting throughput barrier-related C++ preprocessor macros and conditions into `if` conditions. Barrier variants and collector adaptations that are not part of selected throughput-oriented configuration are eliminated, while `develop`-type flags are merged and converted into a single global boolean variable that controls whether throughput-oriented collector mode is enabled. The initial value of the flag is controlled by the JVM configuration at startup. The refactoring does not pose any challenges, because "Raw Parallel" variant barrier and collector adaptations, when disabled, do not conflict with baseline G1 mechanisms and vice versa.

In order to avoid conflicts with the mutator and ensure the ability to adjust the garbage collector state, barrier switch can only happen at safepoint when application threads are paused. The rest of this section focuses on the second part of the implementation

3.4.1 Barrier switch technique

In the HotSpot virtual machine, barrier code generation is independently implemented for the interpreter, and the C1 and C2 compilers. HotSpot generates the whole interpreter machine code and stubs (e.g. `arraycopy` routines) at virtual machine initialization time, while both C1 and C2 compilers generate code on method level concurrently to the running application. All compiled code embeds the write barriers. When barriers are switched, already generated machine code has to be updated or replaced. This poses the main obstacle in the barrier switching implementation. There are several potential approaches to the implementation of barrier switch:

1. The simplest approach to integration of baseline and "Raw Parallel" barriers is combining them into a single barrier and using a global flag to select barrier variant at run time. While this approach has the benefit of avoiding recompilation at barrier switch, it also introduces an extra top-level conditional branch and, as a consequence, throughput penalty for both baseline and "Raw Parallel" barriers, which persists throughout the virtual machine run regardless of barrier switch frequency. Given the primary goal of this thesis, such solution is not considered optimal.

2. Another way of barrier switch involves patching barriers in compiled code. While this approach avoids continuous runtime costs, it is highly non-trivial for implementation — all barrier occurrences in compiled code need to be identified and overwritten. This is complicated by the fact that the exact form of the barrier (i.e. register allocation, inline parts) might differ depending on the context. Outright complexity of this approach makes it unfeasible in presence of other options.

3. In terms of continuous overhead and implementation complexity, the most adequate option is full recompilation on the barrier switch. Using the existing HotSpot mechanisms, previously compiled code can be easily discarded, and it will be later recompiled with a new barrier. Even though considerable costs of recompilation can be expected, the penalty is only incurred at point when the barrier switch happens: recompiled code efficiently integrates different barrier variant, inducing no runtime penalty once the recompilation is done.

The balance of implementation simplicity and prevention of continuous throughput penalty makes the last option — full recompilation — the most practical in the scope of this thesis.

```
1 CodeCache::mark_all_nmethods_for_deoptimization();
2 Deoptimization::deoptimize_all_marked();
```

Listing 10: Deoptimization (discarding) all JIT-compiled code in HotSpot

Listing 10 demonstrates deoptimization of all previously compiled code, which discards all previously compiled code and is executed when the barrier switch happens.

While conceptually simple, this approach requires more adjustments than that in order to make it function correctly within HotSpot. The primary obstacle is the fact that HotSpot JIT-compilation threads are running concurrently even when mutator threads are paused at a safepoint and garbage collection is on-going. This produces a race condition: if some method is being compiled at the same time as barrier switching is induced, the compilation will not be aborted automatically, and as a result it may contain the previous version of the barrier or even a mix of different barriers. Baseline and throughput barriers provide different set of guarantees to the collector, so such a mix of barriers will lead to incorrect behavior when the collector expects stronger guarantees (baseline mode) than the barrier provides (throughput mode). The thesis introduces a special atomic barrier epoch counter for every compilation. This counter is incremented on every barrier switch, thus making barrier switch easy to detect. The JIT compilers save the value of the counter at the beginning of the method compilation, use the saved value to determine current barrier mode throughout the compilation, and compare the epochs at the end. If the epoch values differ, barrier switching has occurred in meantime and compiled code is discarded.

Furthermore, full deoptimization of barrier switching code is only possible for code produced by the JIT-compilers. Machine code generated for the template interpreter and code stubs is not affected by deoptimization and not recompiled. Within the current architecture of HotSpot, these parts of code cannot be switched or altered after the initialization stage. Changes that would enable switch between different barrier versions require considerable redesign — the template interpreter and stubs need to be decoupled from the global state, and several alternative versions of interpreter and stubs need to be instantiated in order to switch them dynamically. Such deep refactoring is beyond the scope of this thesis. That problem is mitigated by one-sided compatibility between baseline and throughput barriers: the baseline barrier provides stronger guarantees to the collector because its algorithm subsumes the “Raw Parallel” barrier, thus it can be used even when the collector is in throughput mode (however, the opposite is not true). Therefore, if the template interpreter and stub code is initially compiled for baseline G1, partial barrier switching is still possible for JIT-compiled code, and the collector is guaranteed to work correctly. If the virtual machine were started with the G1 collector in throughput mode, dynamic barrier switch would not be possible, because the interpreter

initially compiled with "Raw Parallel" barrier variant does not satisfy requirements of the baseline G1.

This described issue of partial barrier switching is the main limitation of the current implementation. The impact of using the baseline post-write barrier for interpreted code is alleviated by the fact that HotSpot compiles frequently used methods, so the majority of commonly executed code can benefit from the throughput barrier. At the same time, combined with the performance penalty of deoptimization, such implementation is expected to perform worse than statically-configured "Raw Parallel" barrier.

3.4.2 Garbage collector adjustments

Due to differences between baseline G1 and throughput mode behavior, special adaptations are necessary in the G1 collector to support the barrier switch. Primary divergence between collector variants is handling of the card table. While baseline G1 uses the card table as an intermediate store for dirty cards and accompanies it with dirty card queues and refinement, in throughput mode the card table is considered to be the essential location of dirty cards, and the remembered set maintenance is limited to the concurrent marking process. Moreover, G1 uses special young generation card marking on the card table, which is omitted by "Raw Parallel" variant.

In order to overcome the discrepancy, a card table fixup procedure needs to be introduced as part of barrier switching. At the time of switching, the cards of the young generation need to be re-marked accordingly. In addition, a special flag is introduced to force full card table scan at the next evacuation in case of switch from throughput barrier to baseline. Full card table scan is done only once after the switch and is necessary to identify all dirty cards on the table that are missing from dirty card queues, as "Raw Parallel" barrier does not update those.

With that fixup procedure in place, the G1 collector is capable of correctly handling dynamic barrier switching. However, for optimal performance, another aspect of the switch which needs to be considered, that is concurrent refinement. While shutdown of refinement threads on switch to throughput mode is possible, it is not the most favorable approach due to partial nature of barrier switching. With no concurrent refinement available, interpreted code running the baseline post-write barrier would be forced to perform in-place refinement which would additionally degrade its performance. At the same time, concurrent refinement as used with the baseline barriers is not fully compatible with the "Raw Parallel" barrier. Due to combination of baseline and throughput barriers present in the running application, concurrent refinement adaptation described in the section 3.2.2 is not possible — it conflicts with the baseline post-write barrier.

In order to establish compatibility between the "Raw Parallel" barrier and concurrent refinement, the barrier needs to ensure strict store operation order. For that purpose, card marking operation needs to be preceded by *store-store* barrier, or equivalently assigned *release* semantics. On the Intel x86_64 architecture — the primary implementation target in the scope of this thesis — such adaptation has no extra costs, because the architecture already guarantees that effects of store operations appear in program order [24]. Thus, in this particular case, the presence of store-store barrier addition is entirely theoretical and shall not affect the performance compared to the statically-configured "Raw Parallel" variant. On weakly-ordered processor architectures, such as ARM, store operation order is not defined and the memory barrier may impose additional penalty [33].

The changes described above are necessary only if virtual machine had been started with G1 in baseline mode and barrier switching is enabled. If G1 is initialized in through-

put mode, it is unable to switch into baseline behavior, thus none of the adaptations are necessary.

3.4.3 Barrier switching policy

The description of dynamic barrier switching implementation in previous sections provides details on the implementation of the switching mechanism, however it omits policies on when the barrier switching shall happen. There are several available options of initiating the switching, which include virtual machine configuration flags, an automatic heuristic and a programmatic interface.

A command line flag (`G1ThroughputBarrier`) is used to specify the initial mode of G1 operation and barrier switch policy. The flag is an integral enumeration with three permitted values:

0. G1 starts in baseline mode and automatic barrier switch is not enabled. Programmatic barrier switch is possible.
1. G1 starts in baseline mode and automatic barrier switch is enabled; programmatic switch is permitted.
2. G1 starts in throughput mode and neither automatic, nor programmatic barrier switch is possible.

The programmatic interface is provided to control current mode and automatic mode switch from running application. The interface consists of a single function, declared as shown in listing 11. A programmatic switch is only possible when the virtual machine was started with G1 in baseline mode.

```
1 public class Runtime {
2     /* ... */
3     public native void gcThroughputBarrier(boolean enable, boolean freeze);
4     /* ... */
5 }
```

Listing 11: Java method for barrier switch at run time

Parameters of `gcThroughputBarrier` have following semantics:

- `enable` — controls whether the throughput barrier needs to be enabled at the moment of call.
- `freeze` — controls whether automatic heuristic-guided switch of the barrier mode is permitted in future.

Automatic switching of the barrier is guided by a heuristic. Over time, G1 collects various statistical data that is used to predict and plan future collector behavior. The same mechanism is used to estimate pause time impact of the "Raw Parallel" barrier, thus assessing feasibility of throughput barrier for specific application runtime profile, and take a decision on switch, which happens at the end of collection pause. Section 4.5 provides details on heuristic rationale, design and evaluation.

Chapter 4

Evaluation

This chapter defines the performance measurement methodology used to evaluate the throughput-focused garbage collector variants, introduces the benchmark suite and describes hardware and software setup for evaluation. Afterwards, it presents evaluation results and their analysis and interpretation. Finally, based on the analysis results, a heuristic for dynamic barrier switch is devised and also evaluated.

Validation

Prior to evaluation of the throughput-oriented collector changes from a throughput and pause time impact perspective, the changes have been validated using the HotSpot test suite to ensure correctness of the new garbage collector behavior. Relevant test groups from the HotSpot test suite are `tier1_gc_1`, `GCBasher`, as well as `compiler/arraycopy`. These tests were executed with pre- and post-garbage collection verification enabled in both release and debug build variants — debug builds contain additional assertions that assist in finding broken invariants. No issues were found in the throughput-oriented changes during the validation process.

4.1 Methodology

Consistent evaluation methodology is of paramount importance for meaningful benchmark measurements and their interpretation. This section discusses known approaches of rigorous throughput evaluation in the context of write barriers and Java Virtual Machine, and defines a methodology which is the most applicable with respect to the goals of this thesis. Methodology of measurements consists of several aspects.

4.1.1 Measurement mechanism & scope

The first important aspect of measurement methodology is a mechanism and scope of measurements. This aspect defines, how and for which part of the system measurements are performed.

Mechanisms of write barrier evaluation used in existing research vary widely depending on study aims and purpose specifics. Zorn [77] estimates CPU costs of various barrier variants in isolation and applies a trace-driven approach to determine the number of events of interest to calculate the total impact. While such approach is valid to assess costs of barriers alone, it is not applicable within the scope of this thesis: write barrier

variants proposed in the chapter 3 are expected to have a diverse impact on garbage collector throughput and latency on multiple stages — effects on possible optimizations at JIT compilation time, mutator throughput at run time, changes in garbage collector pause times — and trace-based estimation is not a good fit to estimate these effects. The methodology of Hosking et al. [23] is also not fully applicable within the scope of this thesis. Authors of that paper perform instrumentation of the execution environment to obtain measurement results, and combine it with "record and replay"-driven sessions. While such approach is more suitable to evaluate broader scope of impact of barrier changes, it still contradicts the goals and motivation of this thesis. In particular, the main motivation for this thesis is improvement of G1 throughput on application level while keeping the overall throughput-pause time balance. Instrumentation of the virtual machine is undesirable in such case as it introduces overhead on its own, distorting the application-level measurements. Furthermore, given the scope of anticipated impact of barrier changes, precise "record and replay" of system events might not be the most exact approach from practical viewpoint — changes in mutator throughput might affect its behavior and interaction with the garbage collector, which in turn also affects allocations, heap state, and garbage collector decisions. Such restrictiveness is "unfair" towards the throughput-oriented garbage collector variants, because it impedes G1 ability to drive itself heuristically to achieve optimal behavior. The measurement methodology employed by Blackburn et al. [5] and Yang et al. [74] is also problematic for the same reasons: it involves patches to the virtual machine to obtain metrics of interest, which would not be used in normal in production scenarios, as well as imposes restrictions on adaptive compilation to ensure determinism.

The methodology used in this thesis has to reflect application-level throughput impact of the write barrier changes, combining both effects on throughput of the mutator and changes in garbage collection pause times, so that the influence on real-world software can be estimated based on these results. Analysis of any particular aspect of garbage collector behavior in isolation shall be performed based on existing JVM observability mechanisms instead of instrumentation to avoid extra overhead which would skew the results. In addition, determinism of measurements shall be established on benchmark suite and environment level — fixed benchmark and dependency versions using the same execution environment for all measurements and repeating tests for statistical significance. Additional efforts to ensure reproducibility via restricting JIT compilation or using trace-based approaches are unfeasible.

In practice, the approach used within this thesis is based on measuring the run time of a benchmark as a whole (on operating system process level) and calculating throughput based on the run time. If a particular benchmark explicitly produces some throughput score, that score is preferred and regarded as more precise than the process-level run time measurement — benchmarks are capable of filtering preparation and warm-up activities out when producing the score. Logging levels of the Java Virtual Machine are configured to produce detailed logs of garbage collection activities, and details for pause time evaluation are extracted from the logs. No instrumentation is implemented and no restrictions of virtual machine behavior are applied. While this approach is simpler than the ones used in aforementioned studies, it produces results closer to the real-world usage scenarios of the G1 collector without inducing excess overhead and covers a wide range of the write barrier effects.

4.1.2 Configuration

The second important aspect of a measurement methodology is configuration of the garbage collector, Java Virtual Machine, operating system and hardware. It is crucial to ensure identical configuration for all benchmarking rounds to obtain comparable measurement results. Furthermore, incorrect configuration of the garbage collector is known to lead to suboptimal performance.

One of critical parameters in garbage collector configuration is heap sizing. The study conducted by Lengauer et al. [30] characterizes behavior of some of popular JVM performance benchmark suites. The authors of that paper determine live heap size for each benchmark by trial and error searching for the lowest maximum heap size with which the benchmark executes successfully (without `OutOfMemoryError` exceptions), and use that live size as a baseline to determine realistic heap size. However, such an approach is not optimal for the G1 collector. Results presented in that paper show that for some benchmarks G1 performs full garbage collections, which, normally, it shall not resort to. Configurations of G1 where full collections occur are considered inadequate, and the documentation for G1 [50] has a dedicated section with recommendations on avoiding full garbage collections. Therefore, while taking the minimal live heap size as a baseline for the realistic heap size is possible, it does not represent "healthy" G1 configurations.

Instead, for each benchmark the minimum heap size which does not introduce full garbage collections for G1 is taken as a baseline for realistic heap size, which is then moderately scaled up (presence of full collections is probabilistic with respect to the heap size). At the same time, excessively increasing heap size, if a benchmark is not capable to utilize that, is also harmful: it unrealistically decreases load on the garbage collector, potentially up to the point where very few garbage collections happen and the young generation regions are large enough to hold most of the heap throughout the benchmark run time. Such conditions are particularly unfit for write barrier evaluation — concentration of objects within the young generation favors barrier fast-paths and does not expose shortcomings of the barrier.

Other important aspects of the G1 collector configuration that are explicitly set for benchmarks within the scope of this thesis include:

- Minimum heap size is set to be equal to the maximum heap size, thus producing fixed-size heaps. This minimizes the impact of heap resizing [50] during benchmark runs, thus facilitating quick convergence to steady state of the garbage collector.
- Use of large memory pages by JVM reduces the impact of virtual memory management on program performance by maximizing the area of memory mappings held by the hardware Translation-Lookaside Buffer [51].
- Pre-touch of heap memory pages at virtual machine startup forces operating system to allocate all comitted virtual memory, thus minimizing the impact of the virtual memory subsystem implementation on the benchmark performance [50].
- Debug-level asynchronous logging into a file is enabled without log rotation. This way, a consistent detailed trace of the garbage collector activities is obtained without introducing overhead of blocking log operations [53].

The rest of virtual machine configuration options remain default. With respect to the operating system configuration, memory paging shall be disabled: use of secondary storage

(page file or swap partition) is detrimental for memory access latency, creating unpredictable delays. Consequently, the hardware setup shall have a sufficient amount of main memory. In addition, large memory page support must be enabled within the operating system. From a hardware configuration perspective, use of a fixed CPU frequency is desirable in order to avoid arbitrary effects of processor frequency scaling on the throughput.

4.1.3 Statistically rigorous measurements

Statistically rigorous approach to measurement organization and data analysis is the final aspect of benchmarking methodology. Failure to apply a consistent approach grounded in statistics can lead to misinterpretation of measurement data and wrong conclusions. This section summarizes the application of the methodology defined by Georges et al. [18] in the context of this thesis.

Measurement errors are classified into systematic and random. Systematic errors stem from mistakes in experiments and procedures. Sections 4.1.1 and 4.1.2 define measurement mechanism and configuration aimed at minimizing the probability of biases and systematic errors. Random errors are non-deterministic and unbiased, and their effects are minimized by applying the following statistical model.

For each throughput benchmark, multiple measurements are taken and their results are used to calculate a mean value and a confidence interval. Due to the fact that number of samples taken for each benchmark is small (less than 30), *Student's t*-distribution is used to calculate the confidence interval. Thus, formulas used for statistical analysis of the throughput benchmark measurements are the following:

- Mean value:
$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$
where x_i is i-th measurement result and n is the total number of measurements.

- Standard deviation:
$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}}$$

- Confidence interval:
$$c_{1,2} = \overline{x} \pm t_{1-\alpha/2;n-1} \frac{s}{\sqrt{n}}$$
where $t_{\alpha;n}$ is the value of Student's distribution with α confidence level and n degrees of freedom. For purposes of this thesis, 95% confidence level is considered sufficient.

Comparison of benchmark results for different barrier variants is performed by examining the calculated confidence intervals: if two confidence intervals overlap, the difference between variants is not statistically significant, otherwise mean values are used for comparison.

Procedure for repeated measurements should take into account the nature of a benchmark. Application start-up performance measurement shall involve multiple independent virtual machine invocations, because throughput of start-up phase is substantially affected by initialization activities, class loading, JIT compilation. Performance of long running applications is less affected by the virtual machine initialization costs, thus benchmarks quantifying steady state throughput shall perform several measurement rounds within the same virtual machine invocation, including a set of warm-up rounds. Still, multiple

repeated virtual machine invocations are necessary for steady state benchmarks in order to eliminate non-deterministic effects of execution environment. Geometric mean is used to calculate an aggregated result of multiple benchmark runs.

Pause time analysis has specifics which need to be mentioned separately. Georges et al. [18] focus their methodology on the performance quantification. However, for purposes of this thesis, determining pause time impact of write barrier variants is also important. Notable difference between the throughput and garbage collection pause times measurements is that a benchmark run produces only a single throughput measurement result, whereas number of garbage collection pauses is typically substantial even for short running benchmarks. Therefore, instead of making assumptions about underlying statistical distribution based on a handful of measurements, as it is done for throughput, the large enough statistical population of pause time samples is analyzed directly. As mentioned in the section 4.1.1, pause times should be obtained from garbage collection logs without any special modifications of the virtual machine, thus the same set of benchmark runs can be used to acquire both throughput and pause time measurement results.

4.2 Benchmark suite

This section describes the benchmark suite used for the throughput write barrier evaluation, including setup and JVM flags of individual benchmarks. Benchmark setup and execution is automated via runner scripts available at [60]. In accordance with section 4.1.2, all benchmarks are executed with the following JVM flags (referred as the *default* in this section): `-XX:+UseLargePages -XX:+AlwaysPreTouch -Xms2G -Xmx2G` — 2 GB heaps were determined to be suitable for most cases, however several benchmarks use very specific heap sizes, overriding the defaults. The list below describes individual benchmarks and their setup.

1. WriteBarrier microbenchmark [57] is a part of HotSpot microbenchmark suite. The microbenchmark focuses on the write barrier throughput, and thus it is particularly interesting in the scope of this thesis. It is executed using Java Microbenchmark Harness (JMH) [39]. No extra configuration is performed, the throughput is measured and reported by the harness.

2. CompilerSpeed [65] is a benchmark that measures the throughput of `javac` compiler running in multiple threads in terms of *compilations per second*. The benchmark is executed with the default JVM flags, in three configurations: running for 15, 60 and 180 seconds. In all configurations the benchmark uses the same number of threads which corresponds to the number of physical CPU cores available on the machine. The benchmark is managing warm-up rounds automatically and starts measurement after the warm-up phase. This benchmark are particularly interesting, because the throughput differences of baseline G1 compared to Parallel GC are prominent in this benchmark [65].

3. DelayInducer [20] is a microbenchmark that measures application run time needed for intensive modifications of Java `ArrayList`. The throughput is calculated as a reciprocal of the benchmark run time. This microbenchmark overrides the default heap size, using 3 GB heap (`-Xms3G -Xmx3G` JVM flags). DelayInducer does not perform warm-up rounds. This benchmark exercises reference modifications in the old generation, stressing the slow paths of the barrier.

4. DaCapo [6] is a well-known benchmark suite used to characterize JVM performance. The benchmark suite is comprised of multiple benchmarks. Several benchmarks are available with different workload sizes. Due to diversity of benchmarks from DaCapo suite, individual setup is used for each benchmark; heap sizes (table 2) are calculated using the methodology described in the section 4.1.2. DaCapo suite uses own test harness, which also manages the number of benchmark repetitions and warm-up rounds.

Benchmark	**Heap size (MB)**	**Notes**
batik	1100	Default workload size, DaCapo version `git+309e1fa`
biojava	600	
eclipse	650	
fop	100	
graphchi	288	
jython	150	
luindex	25	
lusearch	30	
sunflow	120	
xalan	50	
zxing	250	
h2	3200	Large workload, DaCapo version `9.12-MR1-bach`
avrora	50	Huge workload, DaCapo version `9.12-MR1-bach`
pmd	1500	
sunflow	500	

Table 2: DaCapo benchmark setup

5. Renaissance [58] is another well-known benchmark suite widely used to benchmark JVM. Similarly to DaCapo, Renaissance also consists of multiple independent benchmarks and includes own test harness, which manages benchmark execution. Due to diversity of the benchmark suite, specific heap sizes are used for different benchmarks of the suite (table 3) — heap sizes are calculated based on the methodology described in the section 4.1.2.

6. SPECjbb2005 [11] is a benchmark emulating a three-tier client/server system. Scaling unit of SPECjbb2005 benchmark is a *warehouse*, which corresponds to a single thread of execution. The benchmark runs transactions (workload) with a preconfigured number of warehouses and reports the throughput. For purposes of this thesis 1, 2, 4, 6, 8, 12 and 16 warehouses are used.

7. pjbb2005 [3] is a fixed-workload variant of SPECjbb2005 benchmark. For purposes of this thesis, 5 000 000 transaction workload is used with the same numbers of warehouses as in SPECjbb2005.

8. SPECjvm2008 [12] is a benchmark suite which consists of several throughput-focused applications. The benchmark suite includes both startup and steady state benchmarks, and automatically runs a warm-up round for the latter group. The throughput is reported as a *number of operations per unit of time*. This thesis uses a heap size

Benchmark	**Heap size (MB)**
page-rank	1900
future-genetic	100
akka-uct	900
movie-lens	750
scala-doku	100
chi-square	500
fj-kmeans	500
finagle-http	100
reactors	1200
dec-tree	500
naive-bayes	3000
als	600
par-mnemonics	200
scala-kmeans	70
philosophers	60
log-regression	725
gauss-mix	500
mnemonics	160
dotty	150
finagle-chirper	250

Table 3: Renaissance benchmark setup

of 2688 MB heap to run all benchmarks from this suite. Several benchmarks are excluded from runs due to their incompatibility with JDK 20.

9. Optaplanner [13] is an open source AI constraint solver, which includes an extensive benchmarking suite. The benchmarking suite has been used to demonstrate throughput differences between JDK versions and garbage collectors. This thesis uses Optaplanner version `8.10.0.Final`, with all dependencies included for reproducibility. The benchmark produces a throughput measurement in form of a score.

10. Rubykon [64] is a Go-Engine written in Ruby, that provides a performance benchmark suite. Compared to the Parallel Collector, G1 shows throughput deficiencies [64] in this benchmark. The benchmark includes a warm-up round and produces throughput measurements in terms of the *number of operations per unit of time*. The recommended 1500 MB Java heap size is used with this benchmark.

11. BigRamTester [69] simulates an in-memory database cache with a Least-Recently-Used (LRU) item replacement strategy. The benchmark has distinct start-up and operational phases, and can flexibly scale the workload with respect to the number of operations (transactions) and heap size. This thesis uses a modified fixed-workload variant of the benchmark to measure start-up and full benchmark throughput. The throughput is calculated in terms of *transactions per second*. All runs use 18 GB heap, 15 or 16 GB reserved space, and 500 000 000 transactions.

4.3 Hardware & Software setup

All benchmark runs are done with the identical hardware and operating system setup, using the same procedure. After each run of the benchmark suite a reboot is performed to reset the environmental state. The system is dedicated for benchmarking, no unrelated activities are running alongside.

Hardware used for benchmark runs is a *Dell XPS 15 9570* laptop. The laptop is equipped with Intel Core™ i7-8750H CPU with base frequency 2.20 GHz, having 6 physical and 12 virtual cores, 32768 MB of DDR4 2667 MHz RAM and SK Hynix PC401 1 TB NVMe drive. The laptop is continuously connected to the power supply and has a steady connection to a wireless network, which is used for management and maintenance.

The laptop does not have multiple CPU sockets, therefore the `UseCondCardMark` flag is disabled during benchmarking for the "Raw Parallel" G1 collector variant and the Parallel Collector.

Software installed on the laptop is based on Debian Linux 11, last updated on February 14, 2023. The system is installed in a minimal variant without the graphical user interface. Noteworthy package versions are Linux kernel 5.10, GCC 10.2.1 (system compiler), Glibc 2.31. OpenJDK 19.0.1 is installed from an external source and used as the boot JDK [36]. Other OpenJDK build dependencies are installed as required by [36]. The operating system is configured to allocate 12288 2 MB huge pages (24 GB in total) on boot for use in HotSpot. The system has no swap space. Furthermore, the CPU is configured to run in 2.20 GHz fixed-frequency mode to avoid indeterminism introduced by frequency scaling.

The benchmarking cycle includes building all benchmarked OpenJDK variants, executing the benchmark suite, and collecting benchmark results and log files. All OpenJDK builds are made using the system C++ compiler with no special compilation flags, except those flags that were introduced within the scope of this thesis (table 4).

4.4 Analysis & Interpretation

This section presents benchmark results, and provides analysis and interpretation of the throughput and pause time effects of throughput-oriented G1 variants. The benchmark suite has been executed five times for each collector variant. The results are aggregated using the method described in section 4.1.3. Results for four garbage collector variants are presented: the baseline G1 and the Parallel Collector, provided by OpenJDK 20 build 27 (Git commit `d562d3fcbe2`), as well as the "Long" and "Raw Parallel" variants described earlier. All throughput charts show statistically aggregated measurements normalized with respect to the result of baseline G1. Therefore baseline G1 results are always equal to 1 and marked with a thick horizontal line. The x axis on the charts shows individual benchmarks and their groups, whereas the y axis represents the improvement over the baseline G1 performance.

4.4.1 WriteBarrier microbenchmark

Figure 7 shows the write barrier microbenchmark results, including individual microbenchmark measurements and the mean. It can be seen that both throughput-oriented G1 variants improve over the baseline G1, although the Parallel Collector still outperforms

either G1 variant. The improvements are consistent — no degradations appear, and only in a single microbenchmark throughput of the baseline G1 and the "Long" variant are the same. Numeric results of the microbenchmark performance are available in table 11.

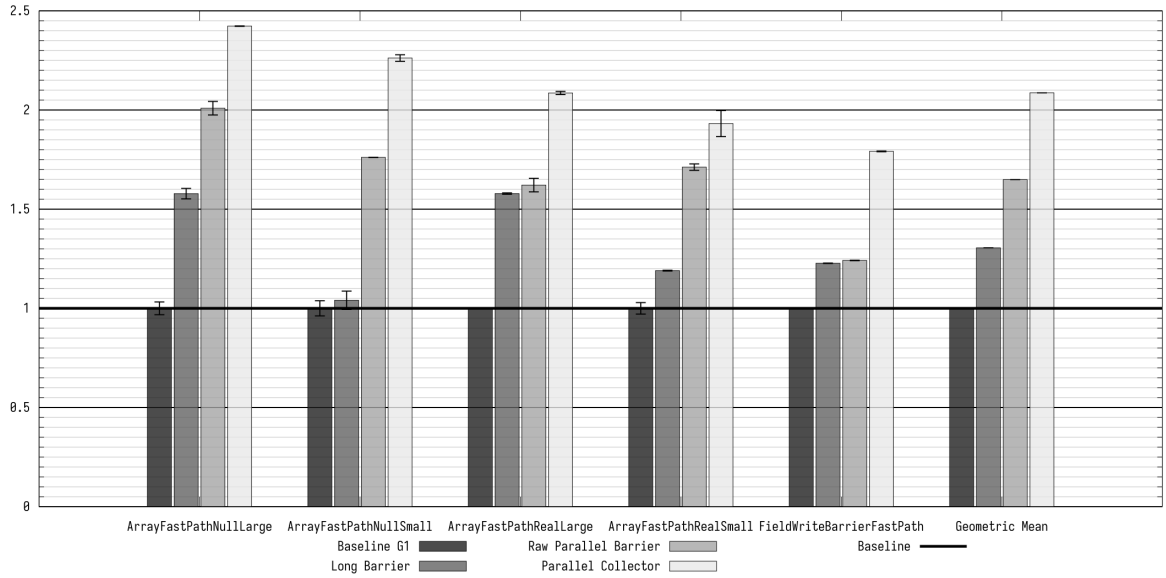


Figure 7: WriteBarrier microbenchmark results

Performance of the throughput write barrier variants in the WriteBarrier microbenchmark is particularly interesting, because this benchmark focuses specifically on the impact of write barriers. Gains produced by the "Raw Parallel" variant are substantial and range between 24% and 100% over the baseline G1, mean throughput gain is 65%. Performance improvement of the "Long" variant is lower with maximum gain over baseline G1 being 58% and mean improvement of 31%. Performance gains produced by the throughput-oriented G1 variants in this microbenchmark can also be treated as a soft upper bound for application-level throughput improvement. For most other workloads the throughput increase produced by alternative write barriers will be partially offset by the costs of garbage collector adaptations. Nevertheless, as shown in the section 4.4.2, certain benchmarks are exceptionally favorable for specific G1 collector configurations, and the total throughput improvement for those surpasses the microbenchmark. In general, this pattern in throughput differences can be observed also in application-level benchmarks, although at a different scale.

Scale of throughput improvements varies across specific WriteBarrier microbenchmark measurements. For the "Raw Parallel" variant and Parallel Collector, the most significant improvements are derived in the null-filled array benchmarks: each array microbenchmark consists of a single loop with a short body, which benefits from the lack of conditional branches and shorter machine code of these write barrier variants. The "Long" barrier demonstrates throughput improvements only in the long variant of the null-filled array microbenchmark, whereas during short runs its behavior is effectively the same as the baseline G1 — their fast-path sections are the same. Nonetheless, the "Long" barrier throughput improves in the "real" array benchmarks: due to presence of non-null references, the complete write barrier has to be executed several times, and the write barrier of the "Long" variant is considerably shorter than the baseline G1 barrier. In the field modification microbenchmark the performance gains of both "Long" and "Raw Parallel" barrier are more moderate. This microbenchmark is very short, with only 4 reference

assignments per single run.

These results show that the short unconditional write barrier such as the "Raw Parallel" barrier (section 3.1.2) is able to provide ample throughput gains. Adding extra pre-filtering conditions to it leads to less optimal performance. The "Long" write barrier variant (section 3.1.3) contains 3 pre-filtering conditions on top of the minimal card marking barrier, and its performance gain is already twice lower on average. At the same time, the "Long" variant results reveal that activities dedicated to the concurrent remembered set maintenance can take up to 25% of the barrier run time.

Simultaneously, it is evident that optimizations implemented in the throughput-oriented G1 collector variants are not sufficient to reach performance of the Parallel Collector. In the WriteBarrier microbenchmark, the Parallel Collector consistently performs better than any of G1 variants, and mean performance difference between it and the baseline G1 is 109%. Despite the fact that the post-write barrier portion of the "Raw Parallel" variant is identical to the write barrier of the Parallel Collector, the latter still performs almost 27% better. This difference can be attributed to pre-write barrier of the G1 collector that is preserved in both throughput barrier variants. Result of a special OpenJDK build with the pre-write barrier removed support this conclusion, showing identical performance of the "Raw Parallel" G1 variant and the Parallel Collector in the microbenchmark.

4.4.2 Application-level throughput

Figure 8 (table 6) shows throughput benchmark results for the rest of the benchmark suite. Overall, the improvement produced by the throughput-oriented G1 variants is evident in application-level throughput too, albeit to a smaller scale. The "Raw Parallel" variant performs significantly better than "Long", and prominent gains are produced for the CompilerSpeed, DelayInducer, Optaplanner, Rubykon and BigRamTester benchmarks. Furthermore, the throughput-oriented variants do not produce significant performance degradations in any of individual benchmarks. However, in many benchmarks the Parallel Collector still performs better than any G1 variant. Appendix C provides detailed throughput charts of the benchmark suite. Detailed results of benchmark suite performance are available in tables 7 (baseline G1), 8 ("Long" variant), 9 ("Raw Parallel" variant) and 10 (Parallel Collector).

CompilerSpeed benchmark results show that the "Raw Parallel" collector variant outperforms all alternatives, producing mean improvement of 7.8% over the baseline G1, while performance of the "Long" variant is not significantly different from the baseline. Mean throughput of the Parallel Collector is only slightly (1.9%) better than the baseline. Detailed inspection of results of specific CompilerSpeed configurations (figure 9) reveals that the baseline G1 and "Long" variant relative performance is consistent and changes very little regardless of benchmark run length. The "Raw Parallel" variant shows limited decrease of throughput in longer runs, going from 9.9% improvement over the baseline for 15 second runs to 6.5% for 180 second runs. However it still performs significantly better than other G1 variants. The Parallel Collector performs 6.9% better than the baseline G1 in short 15 second run, but during longer runs its throughput drops below the baseline level. The main reason for that are longer garbage collection pause times of the Parallel Collector. Total pause time of benchmarked G1 variants is $6.7-7\%$ of the benchmark run time, whereas for the Parallel Collector this number is 18.4%, which negates any positive impact of the write barrier on application-level throughput. The results of CompilerSpeed benchmark demonstrate that, while minimal unconditional card marking barrier

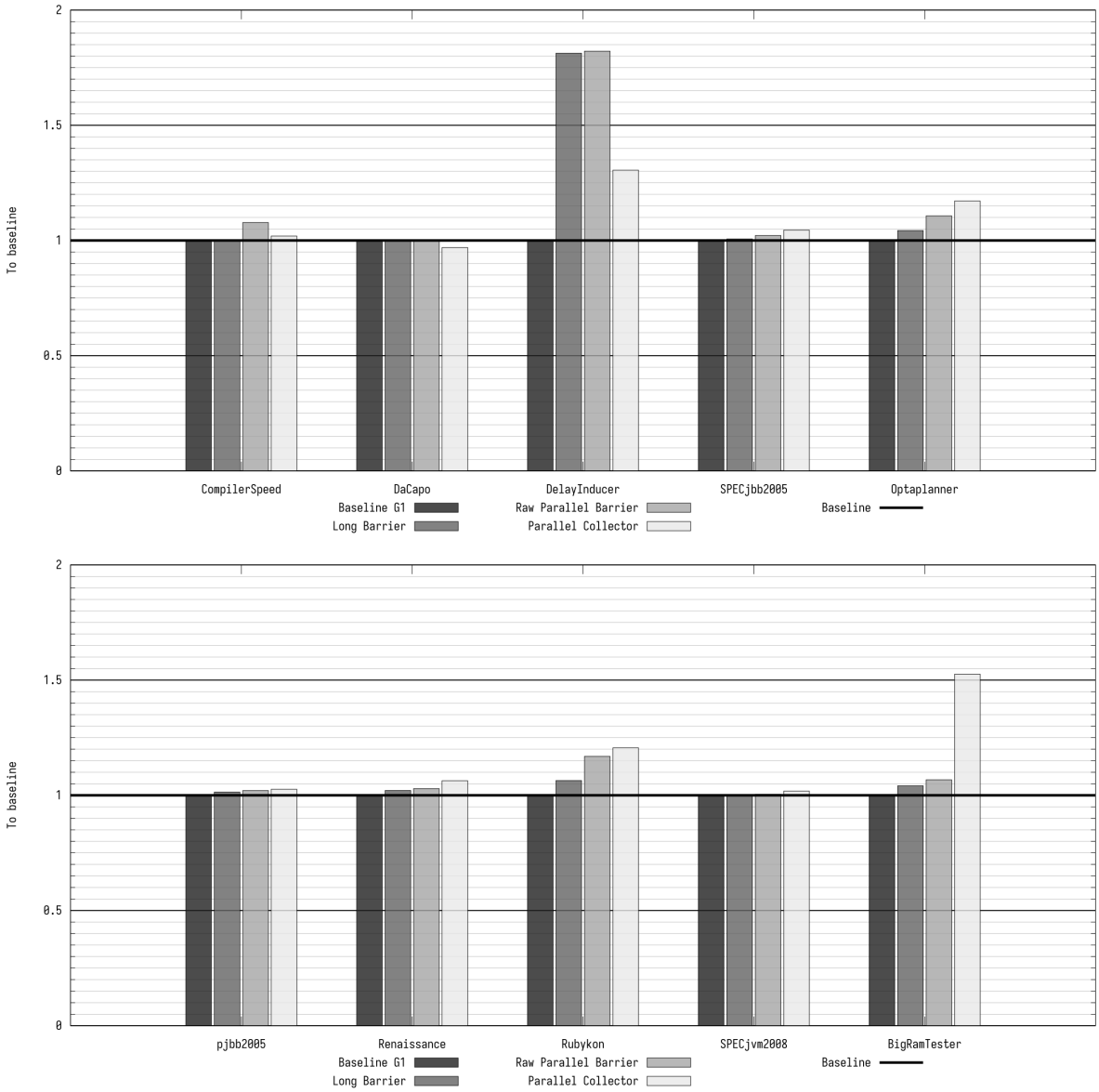


Figure 8: Overview of benchmark suite results

is beneficial for throughput and produces meaningful gains, other aspects of the garbage collector behavior may negate these gains in the long run.

DaCapo benchmark shows little to no differences between the baseline G1 and its throughput-oriented variants in overwhelming majority of individual benchmarks (figures 10, 11 and 12). In addition, in some cases high variance of benchmark results produces wide confidence intervals that overlap with any potential realistic gains, thus making the observed differences statistically meaningless. The Parallel Collector exhibits slightly different throughput results. In two benchmarks (lusearch, xalan) the Parallel Collector performs significantly worse than any of G1 variants — the reasons for that include differences between G1 and the Parallel Collector young generation sizing policy. These two benchmarks distort mean results for the whole benchmark suite. However, in most other cases differences between G1 and the Parallel Collector are minor, showing little difference between garbage collectors. Thus, the DaCapo benchmark suite is not sensitive enough to

the barrier changes.

DelayInducer benchmark results show (figure 13) outstanding gains produced by the throughput-oriented G1 variants: The "Long" and "Raw Parallel" variants outperform the baseline G1 by 81% and 82% respectively, also surpassing the Parallel Collector, whose throughput is 31% above the baseline. These throughput improvements are not only higher than for any other benchmark, they exceed the mean throughput-oriented barrier variant performance in the WriteBarrier microbenchmark. The reason for this improvement is due to the benchmark itself: it intensively modifies contents of large Java `ArrayList` objects that contain non-repeating long-living `Integer` objects. Such behavior leads to the heap modifications pattern where most modified references are between the old generation objects, thus invoking the execution of G1 post-write barrier slow-path (refinement operations in the baseline G1). These conditions are particularly unfavorable for the baseline G1 write barrier, but benefit the more lightweight "Long" and "Raw Parallel" barriers. At the same time, impact of the write barrier is not the only component of observed throughput gains, otherwise the Parallel Collector should have performed at least as well as the "Raw Parallel" variant. Inspection of the garbage collection pauses showed that the pause times of the Parallel Collector are much longer on average than those of G1, which diminishes the difference in application-level throughput.

SPECjbb2005 benchmark shows mild differences between different G1 variants and the Parallel Collector. Overall, throughput gains increase the simpler the barrier is: there is no statistically significant improvement over the baseline G1 for the "Long" variant, 2.1% for the "Raw Parallel", and 4.5% for the Parallel Collector. Throughput differences in individual SPECjbb2005 configurations (figure 14) do not show a significant dependence on number of warehouses in the configuration. Overall, while the pattern of throughput results conforms with the initial expectations, scale of changes is minor.

Optaplanner benchmark results show considerable gains in mean throughput, whose pattern matches prior expectations: The "Long" variant performs 4.3% better than the baseline G1, the "Raw Parallel" variant improves over the baseline by 10.7%, while the Parallel Collector results are 17.1% better than the baseline. Breakdown of individual Optaplanner benchmark throughput results (figure 17) shows the same pattern for most individual benchmarks, although with some differences in scale of improvements. Only in two cases there are no statistically significant differences between G1 variants. The results of Optaplanner benchmark are particularly interesting in the scope of this thesis, because this benchmark has been previously used to demonstrate [13] deficiencies in G1 throughput. Obtained benchmark results show that throughput-oriented collector variants partially mitigate this issue.

pjbb2005 benchmark produces results that are similar to SPECjbb2005, which is the expected behavior — pjbb2005 is a fixed-workload variant of SPECjbb2005 benchmark. Individual pjbb2005 configurations (figure 15) show that low-warehouse runs do not produce meaningful measurement results due to lower overall load — number of transactions (workload) is fixed per warehouse, whereas the results of other runs are close to those of SPECjbb2005 within confidence intervals.

Renaissance benchmark shows moderate mean throughput gains for the throughput-oriented G1 variants, which are however somewhat lower than the results of the Parallel Collec-

tor. Renaissance is a diverse benchmark suite, individual benchmark results (figure 18) are more interesting. In many benchmarks there is no statistically significant difference between G1 collector variants, sometimes due substantial variance. In two cases, contrary to the expectations, the "Long" variant outperforms "Raw Parallel". Throughput gains produced by the Parallel Collector are more consistent. Similarly to DaCapo, the Renaissance benchmark suite does not reveal significant differences between G1 variants, which means that for most benchmarks in the suite the write barrier costs are minimal. Possible reasons of that include object concentration in young generation regions, which favors the write barrier fast-path, and thus does not reveal full costs of the baseline G1 write barrier.

Rubykon benchmark results demonstrate substantial throughput improvements produced by the throughput-oriented G1 collector variants. In particular, the "Raw Parallel" variant performs 16.9% better than the baseline G1, getting close to the Parallel Collector performance (20.6% over the baseline). Gains produced by the "Long" variant are more modest — 6.4% over the baseline. The results are consistent and have very little variability (figure 16). Rubykon is one of the benchmarks where baseline G1 throughput issues were observed before [64], and obtained results demonstrate that throughput-oriented collector variants are capable to largely mitigate the problem.

SPECjvm2008 benchmark shows little to no difference between differed G1 collector variants and the Parallel Collector. The SPECjvm2008 is a benchmark suite which consists of multiple independent workloads. Results of individual benchmarks (figures 19 and 20) show that in the majority of cases there is no statistically significant difference between any G1 variant and the Parallel Collector. The variability of results sometimes is large enough to make any realistic performance gains statistically meaningless. Overall, the SPECjvm2008 benchmark suite also does not demonstrate enough sensitivity to write barrier changes.

BigRamTester benchmark demonstrates moderate throughput gains produced by the throughput-oriented G1 collector variants, 4.1% and 6.7% over the baseline G1 for the "Long" and "Raw Parallel" variants respectively. These results are far surpassed by the Parallel Collector, which performs 52.6% better that the baseline. Such divergence is not caused solely by the write barrier impact on throughput, as shown by BigRamTester performance decomposition into mutator throughput and garbage collector pause time on the figure 21. The mutator throughput changes, which are directly influenced by the write barrier, follow the expected pattern: the "Long" variant results are 11% better than the baseline G1, the "Raw Parallel" variant performs 18.9% better than the baseline, and difference between the Parallel Collector and the baseline G1 is 28%. However, despite clear improvements in mutator throughput, the garbage collection pause times distort the application-level results.

The BigRamTester benchmark is especially advantageous for the Parallel Collector, whose garbage collector pause times in total are only $\frac{1}{4}$ of the baseline G1 pause times. For throughput-oriented G1 variants the pause times are expectedly longer, comparing to the baseline G1, being 14.9% and 24.4% longer for the "Long" and "Raw Parallel" variants respectively. Therefore, application-level throughput of the benchmark is proportionally higher than mutator-only throughput for the Parallel Collector and lower for the throughput-oriented G1 variants.

Overall, the application-level throughput results produced by the "Long" and "Raw Parallel" variants show significant improvements, especially, in throughput-oriented workloads. Previously reported throughput issues with Optaplanner, Rubykon, CompilerSpeed benchmarks are largely mitigated by the throughput-oriented G1 variants. At the same time, DaCapo, Renaissance and SPECjvm2008 — well-known JVM benchmark suites — did not show any meaningful differences between different G1 variants. Furthermore, as evidenced by BigRamTester results, the impact of changes in the write barrier may also be offset by pause time effects of the garbage collection adaptations, thus diminishing possible gains. Nevertheless, throughput-oriented collector variants still often unable to reach performance of the Parallel Collector. Section 4.4.3 discusses the impact of changes on the garbage collection pause time in more details.

4.4.3 Pause time analysis

Garbage collection pause time impact of the G1 collector adaptations is important to characterize in order to understand trade-offs offered by the new throughput-oriented collector variants. Longer pause times are expected primarily due to the fact that throughput-oriented G1 variants perform full card table scan. Longer garbage collection pauses increase the latency of application responses to incoming requests, which is crucial in certain classes of workloads. In addition, as demonstrated in section 4.4.2, substantial differences in the garbage collection pause times are also a factor of application-level throughput changes. This section focuses on the benchmarks which demonstrated noticeable differences in throughput. Comparison is being done among the baseline G1, "Long" and "Raw Parallel" collector variants — the Parallel Collector is omitted as its algorithm and garbage collection pause structure are too different from G1 to be meaningfully correlated with it.

Results are obtained from the same set of benchmark runs which were used for throughput analysis, therefore their evaluation presented below directly corresponds to throughput results in the section 4.4.2. For each benchmark, garbage collection pause information is extracted from G1 logs, pauses related to concurrent marking cycle are filtered out as irrelevant in the context of this thesis, and the rest are presented as a box-and-whisker plot; whisker part is calculated based on *interquartile range (IQR)* as $1.5{*}IQR = 1.5{*}(Q_3{-}Q_1)$. The plots present actual pause times and also include maximum and minimum pause time values. The x axis on all plots represents individual benchmarks or benchmark groups, the y axis shows garbage collection pause times in milliseconds. Corresponding pause time values are available in tables 13, 14 and 15.

CompilerSpeed benchmark has been executed in three configurations: for 15, 60 and 180 seconds. Figure 22 presents measured garbage collection pause times for the benchmark. Median pause times exhibit at most few millisecond ($\approx 2.6\%$) difference among garbage collector variants. The overall distribution of pause times is also slightly wider for throughput-oriented G1 variants, which means that there is marginally more variance in garbage collection pauses of "Long" and "Raw Parallel" variants. In CompilerSpeed benchmark, the maximum pause time values are inconsistent across runs, and thus are not taken into account.

DelayInducer benchmark produces garbage collection pause time distribution with larger variance for the throughput-oriented collector variants. For "Long" and "Raw Parallel" variants the upper bound of the distribution is respectively 4 and 8 milliseconds higher

than for the baseline G1. At the same time, distribution lower bounds for the throughput-oriented collector variants are much lower than for the baseline G1, which also slightly (2 milliseconds) lowers the median value. This behavior is the result of massive differences in throughput between the baseline G1 and throughput-oriented variants, which leads to considerably shorter application run times and lower number of performed garbage collections — the outlier values have more influence on the resulting distribution. Overall, the practical impact of throughput-oriented barriers on garbage collection pauses is very limited.

Optaplanner benchmark consists of multiple individual workloads. Garbage collection pause time behavior exhibited by these workloads can be classified into two groups as demonstrated in figure 25. Most of workloads belong to the group `I`: despite present relative differences in the median and overall pause time distribution, absolute change in pause times is not larger than 0.5 millisecond, so in practice the impact of throughput-oriented barriers is marginal. Two of Optaplanner benchmarks belong to the group `II`: due to much longer garbage collections, relative differences between G1 variants are less prominent than in the group `I`. Pause time distribution upper bounds in the group `II` are 5 milliseconds higher for the throughput-oriented collector variants. The outlier maximum values are also $10 - 15$ milliseconds higher, thus the practical impact of throughput-oriented collector variants on group `II` benchmarks is modest.

Table 12 summarizes the impact of throughput-oriented G1 variants on the absolute and relative durations of mutator and pause time throughout the Optaplanner benchmark run. In absolute terms, the differences between G1 variants are very minor due to the fact that Optaplanner benchmark run include preparation and warm-up activities. Relatively, the throughput-oriented barriers increase the fraction of pause time only by $0.03 - 0.05\%$ of the total application pause time. Considering the substantial throughput improvements demonstrated in the section 4.4.2, pause time impact of the throughput barriers in Optaplanner benchmark is negligible.

Rubykon benchmark also shows very limited impact of throughput-oriented collector variants. Median garbage collection pause times of "Long" and "Raw Parallel" variants are 0.5 milliseconds longer than the median of the baseline G1. The upper bound of overall distribution of pause times is also $2 - 2.5$ milliseconds higher. The maximum pause times are show high variance across Rubykon runs with no statistical significance. Even though the relative impact ($< 6\%$ difference in the median values) of throughput-oriented collector variants is clearly visible on the plot, its absolute scale is low due to short garbage collection pauses.

BigRamTester benchmark produces significantly different results (figure 26), compared to the previously described benchmarks. Even for the baseline G1, median pause time is substantial — 480 milliseconds, and overall pause time distribution spans between 475 and 490 milliseconds. Results produced by the throughput-oriented collector variants are not only higher, their distributions (except outliers) do not intersect with the baseline G1 pause times, and thus they are statistically different. The "Long" variant produces pause times between 505 and 555 milliseconds, with a median value below 520 milliseconds, whereas for the "Raw Parallel" barrier the pause time distribution is between 545 and 595 milliseconds and the median above 555 milliseconds – considerably higher than the alternatives. Considering the fact that pause times add up throughout the application

run, such differences not only affect the individual operation latencies, but also diminish the application-level throughput, as has been demonstrated in section 4.4.2.

The BigRamTester benchmark also demonstrates different aspects of throughput-oriented barrier impact on collector pause times. Figure 27 shows individual garbage collection pause times of a single run of the BigRamTester benchmark. The x axis represents the benchmark run time, while the y axis shows pause time lengths in milliseconds. The plot shows that most pause times are clustered by G1 collector variant and their duration within a cluster is roughly equal. The differences in pause time lengths between the clusters are direct consequence of full card table scan and refinement changes in the throughput-oriented collector variants. However, the "Long" and "Raw Parallel" variants also produce sequences of outliers, whose pause times are much higher. Experimentation with BigRamTester workload and heap size revealed that such outliers are a sign of particularly high load on the garbage collector, although they are not full garbage collections yet. The reason for increased load on the collector compared to the baseline G1 is an indirect effect of the throughput write barrier — improved throughput of the mutator creates extra strain on the garbage collector. Overall, the pause time impact of throughput-oriented collector variants on BigRamTester benchmark is particularly interesting, because this benchmark flexibly scales with heap size. 18 GB heap was used in measurements, which is much larger than heaps used for other benchmarks. Therefore, results of BigRamTester are more applicable to real-world workloads with large heaps.

Detailed DaCapo, Renaissance, SPECjbb2005, pjbb2005 and SPECjvm2008 benchmark pause time analysis have been omitted from the above description for conciseness. Tables 13, 14 and 15 summarize pause time behavior for these benchmarks. The tables present mean, median, outlier, upper and lower bounds (based on IQR), and total sum of the pause time distribution for the baseline G1, "Long" and "Raw Parallel" variants, respectively. The results follow the overall trend and show rather moderate impact of throughput-oriented write barriers in the majority of cases. No abnormal behavior has been observed. In fact, the only benchmark where garbage collection pause times significantly differed among different G1 variants is BigRamTester.

Figure 29 (table 16) breaks down the impact of the changes on garbage collection phases of BigRamTester benchmark. Pause time is dominated by the heap root scan and object copy phases. The longest phase of the garbage collection — object copy phase — shows only a minor increase in length for the throughput-oriented collector variants. Much more substantial increase in duration happens in the heap root scan phase, which is the main contributor into longer garbage collection pause times in "Long" and "Raw Parallel" collector variants. In addition, figure 28 shows heap root scan durations for individual garbage collections. Observed behavior closely follows the garbage collection pause time plot (figure 27), confirming the conclusion that throughput-oriented collector variants impact the garbage collection pause times primarily by increasing heap root scan phase duration. The increase is directly proportional to the number of scanned cards.

4.4.4 Other aspects

Chunk table modification as part of the write barrier is an attempt to mitigate the degradation of garbage collection pause times, particularly the heap root scan phase, at the expense of a larger write barrier (section 3.2.3). This approach has not been selected for thorough evaluation, because it does not improve the write barrier throughput, which is the main goal of this thesis. Nevertheless, a brief investigation of it has been performed

to characterize its impact on throughput-pause time trade-off. Figure 30 (table 17) from appendix E shows WriteBarrier microbenchmark results for the "Raw Parallel" variant with and without chunk table modification. It performs substantially worse than the normal "Raw Parallel" barrier in all measurements, and in one case it even performs slightly worse than baseline G1. Even though its mean throughput is still better than the baseline, the throughput improvement is only 29% over the baseline G1, much worse than the "Raw Parallel" barrier without chunk modification (65% over the baseline). At the same time, the reduction in the garbage collection pause times provided by this change is not sufficient to reach the application-level performance of the "Raw Parallel" variant. Figure 31 (table 18) compares pause time, mutator throughput and application throughput of the BigRamTester benchmark for "Raw Parallel" variant with and without chunk dirtying with baseline G1. Application-level throughput of the variant with chunk table modification does not perform as well as the "Raw Parallel" variant — the reduction in garbage collection pause times cannot fully compensate the drop in the mutator throughput.

4.5 Dynamic switch heuristic

This section provides a description and rationale for the write barrier dynamic switch heuristic discussed in section 3.4. A heuristic has been formulated based on the analysis of throughput-oriented barrier pause time impact done in section 4.4.3.

The core idea of dynamic switch heuristic is the estimation of throughput-oriented write barrier impact on garbage collection pause times, performing the switch once the anticipated impact is below certain threshold. As shown during the benchmark suite pause time impact analysis, the pause time impact of throughput-oriented write barrier is often modest, and the switch threshold set accordingly. The main contributor towards longer pause times is the full card table scan, thus the number of dirty cards on the card table can be used as a proxy to estimate potential impact of throughput write barriers. Based on this assumption, the following formulas estimate the possible impact of switching into the throughput mode and determine whether to switch:

$$
\begin{aligned}
C_{baseline} &= C_{rate} \times T_{mutator} + C_{buffers} \\
C_{throughput} &= C_{baseline} \times \frac{R_{total}}{R_{old}} \times \frac{P_{total}}{P_{total} - 0.25 \times P_{same}} \\
T'_{scan} &= C_{throughput} \times \frac{T_{scan}}{C_{scanned}} \\
I_0 &= \frac{T'_{scan} - T_{scan}}{T_{pause}} \\
S &= \begin{cases} 1 & \text{if } Predict(I_0, I_1, ..., I_n) \leq 0.075, \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Where

$C_{baseline}$ is estimated total number of cards dirtied by the mutator in baseline mode.
$C_{throughput}$ is estimated total number of cards dirtied by the mutator in throughput mode.
C_{rate} is predicted card dirtying rate of the mutator in baseline mode.
$C_{buffers}$ is predicted number of dirty cards in thread buffers in baseline mode.

$C_{scanned}$ is the actual number of scanned dirty cards during the last heap root scan phase.
$T_{mutator}$ is the actual duration of the last mutator time slice.
T_{pause} is the actual duration of the last garbage collection pause.
T_{scan} is the actual duration of the last heap root scan phase.
T'_{scan} is estimated duration of heap root scan phase in the throughput mode.
R_{total} is the total number of active heap regions.
R_{old} is the number of old heap regions.
P_{total} is the total number of scanned references.
P_{same} is the number of scanned same-region references.
I_0 is estimated impact of throughput mode on the last garbage collection.
$Predict(I_0, I_1, ..., I_n)$ is predicted throughput mode impact based on estimates done for the n last garbage collections.
S is the decision to switch into throughput mode.

The heuristic based on the throughput mode impact estimation formula depicted above is as follows:

1. Estimation takes place at the end of young and mixed garbage collections, taking into the account statistics collected during the last and previous collections. Estimation only happens in baseline mode, rollback from the throughput to the baseline write barrier uses a separate policy described later.

2. Number of cards dirtied during the last mutator time slice $C_{baseline}$ is estimated using predicted card dirtying rate C_{rate} (G1 already collects that information) and a predicted number of cards in per-thread buffers $C_{buffers}$. This number is taken as a baseline which is used in further computations. In case when predicted number of dirtied cards is below the number of actually scanned cards from the last garbage collection, the latter is taken as a baseline.

3. The baseline number of dirty cards $C_{baseline}$ is up-scaled with two factors:

   - Proportion of old regions among the total number of active heap regions ($\frac{R_{total}}{R_{old}}$) — to account for the absence of young generation region checks in the throughput write barrier.
   - Proportion of same-region references among the total number of references scanned ($\frac{P_{total}}{P_{total} - 0.25 \times P_{same}}$) — to account for the absence of cross-region reference check in the throughput write barrier. Additional statistics collection has been implemented within G1 to gather that information. 0.25 coefficient is used to limit the impact of up-scaling. The coefficient value has been determined empirically.

   The resulting number $C_{throughput}$ is assumed to be a number of cards which would have been dirtied by the throughput write barrier, had it been active during the last mutator time slice.

4. Heap root scan phase duration in throughput mode T'_{scan} is estimated based on the previously computed number of dirty cards $C_{throughput}$ and the actual average card scan rate of the last garbage collection $\frac{T_{scan}}{C_{scanned}}$.

5. Finally, the relative impact of heap root scan phase duration increase is calculated ($\frac{T'_{scan} - T_{scan}}{T_{pause}}$). If the impact has been sustained below the switch threshold for several

garbage collections, the barrier switch activates. The default switch threshold is set to 7.5% and can be configured at virtual machine startup using a JVM flag.

The described heuristic focuses on keeping the impact of the throughput write barrier on the heap root scan phase at a reasonable level based on the data obtained during last garbage collections. Deficiencies of the proposed approach include exclusive focus on a single phase of garbage collection pause, as well as the inability to precisely know the number of dirty cards produced by the mutator and its limited ability to adapt to sudden changes in application behavior. In order to solve the latter, a policy to rollback to the baseline write barrier is devised:

- Barrier rollback happens only when full garbage collection occurs — it is considered a sign of G1 inability to keep up with the mutator activity. With no full garbage collections the G1 collector can stay in throughput mode indefinitely, its state is not reevaluated after switching to the throughput write barrier happens. Full garbage collections caused by `System.gc()` calls are ignored.
- After full garbage collection happens, all statistics related to the throughput barrier switch heuristic are reset to delay any possible switch back to the throughput barrier. Furthermore, any future barrier switches are penalized by multiplying the switch threshold by a factor, thus making it harder to reach. The default penalty factor is 0.9; it can be changed at virtual machine startup via a JVM flag.

Barrier switch and rollback logic is implemented as part of the G1 policy class, where it is able to take advantage of existing G1 statistics collection and prediction functionality.

4.5.1 Dynamic switch evaluation

This section presents the essential results of the dynamically-switched barrier and the switch policy described in the sections 3.4 and 4.5 from a throughput and pause time impact perspective, comparing them to the baseline G1 and the "Raw Parallel" collector variant the dynamic switch is based on. The same methodology, benchmark suite and setup (sections 4.1 to 4.3) are used. In all benchmarks the heuristic triggered at most once once per run with no rollbacks. All benchmarks except BigRamTester have been executed with the default dynamic switch configuration[1]. For BigRamTester, a higher switch threshold of 25% has been used to ensure consistent early switch during BigRamTester startup phase. The figures discussed in this section are located in the appendix F.

Throughput evaluation results for the dynamically-switched barrier are presented on figure 32 (table 19). Overall, the performance of dynamic barrier switch heavily depends on specific benchmark run time. In long-running benchmarks, such as Optaplanner and BigRamTester, throughput achieved by the dynamically-switched barrier reaches results of the "Raw Parallel" collector variant. In Rubykon — a benchmark of medium length — the dynamically switched barrier achieves 3.5% improvement over the baseline G1, which is far worse than the improvement (16.9%) produced by the "Raw Parallel" variant. In many other benchmarks, such as DaCapo, SPECjbb2005, pjbb2005, Renaissance, SPECjvm2008, the difference between the baseline G1 and dynamically-switched barrier is either minuscule, or is not present at all. At the same time, in these benchmarks the gains produced by the "Raw Parallel" variant are also very limited. In short-running

[1] 7.5% switch threshold and 0.9 rollback penalty.

CompilerSpeed and DelayInducer benchmarks the dynamically-switched barrier actually performs worse than the baseline G1, in stark contrast with the "Raw Parallel" barrier that shows substantial throughput increase in these benchmarks.

Shortcomings of the barrier switch implementation are the main reason for throughput degradation in shorter workloads and insignificant throughput gains in some others. The barrier switch happens via deoptimization of all JIT-compiled code, which imposes a considerable throughput penalty. Furthermore, barrier switch is incomplete and the dynamically switched barrier in throughput mode actually runs a mix of the baseline and "Raw Parallel" barriers. The dynamically-switched barrier brings very limited benefit for benchmarks that heavily rely on code paths that still contain the baseline barrier (e.g. array copying). In combination, these factors limit the usefulness of dynamically-switched barrier guided by the heuristic to a certain class of long-running workloads, where the dynamic barrier switch can perform on a par with the "Raw Parallel" variant.

WriteBarrier microbenchmark cannot be executed with the automatic heuristic-guided barrier switch due to the fact that the microbenchmark is short and automatic switch does not happen before the microbenchmark finishes. For purpose of running this microbenchmark, the dynamically switched barrier had been modified to perform an unconditional switch after the first garbage collection. Figure 34 (table 21) shows the throughput results for the switched barrier in that configuration. As expected, the results of barrier switch are equivalent to the "Raw Parallel" variant, which it is based upon. The microbenchmark does not exercise code paths where barrier switch is not possible and performs warm-up rounds which ensure that the C2 compiler is activated and the throughput write barrier is used in hot code paths, thus deficiencies of the barrier switch are not present in WriteBarrier microbenchmark results.

Startup performance is an important use case for dynamic barrier switching. Instead of relying on the heuristic to switch barriers automatically, the application uses the programmatic interface to enable the barrier that is the most favorable in a certain phase. In some application classes, such as Web servers and databases, higher throughput is desired at startup, because it enables quicker transition into the operational phase, where shorter and more predictable garbage collection pauses are preferred over the gains in throughput. Therefore, a special benchmark based on BigRamTester has been implemented to evaluate this use case. The benchmark programmatically switches the barrier to enable the throughput mode at the beginning of the BigRamTester run and rollbacks to the baseline G1 barrier once startup finishes. Startup time is measured and the performance is calculated as a reciprocal of the startup time. This benchmark uses the same heap and workload size as BigRamTester, however it decreases the reserved space to 15 GB to mitigate inherent randomness of BigRamTester startup garbage collections.

Figure 33 (table 20) demonstrates the startup performance improvements produced by the dynamically-switched barrier over the baseline — the dynamically-switched barrier exhibits 6.6% improvement in startup throughput (that is, 6.2% shorter startup times) on the application level. Gains in mutator throughput are even higher — 17.5% better throughput during startup. However, this substantial growth of mutator throughput wanes on application level due to extensive use of `System.gc()`-induced garbage collections that consume considerable amount of startup run time. Note that changes in garbage collection pause times presented on the figure 33 are very minor, because full garbage collections are excluded from the figure and the number of young collections is low at startup time.

Garbage collection pause times of the dynamically-switched barrier show mixed behavior, which is demonstrated on the figure 35 (table 22). Pause time distribution produced by the dynamically-switched barrier in BigRamTester benchmark is located between the "Long" and "Raw Parallel" variants. Use of conditional refinement by the throughput mode of dynamically-switched collector variant reduces its pause times compared to the statically-configured "Raw Parallel" variant which does no refinement. At the same time, due to use of unconditional minimal card marking write barrier, the mutator performance of dynamically-switched collector variant in throughput mode is higher than that of the "Long" variant and no filtration of references is happening, thus performant mutator also produces extra load on the garbage collector. Furthermore, the implementation of dynamic barrier switch places additional conditions and statistics collection on G1 garbage collector hot paths, thus introducing overhead. This additional overhead is apparent when comparing the individual garbage collection pause times of the baseline G1 and the dynamically-switched barrier (figure 36): at the early stages of execution, until the switch happens, the dynamically-switched barrier exhibits slightly longer garbage collection pauses despite using functionally the same algorithm as the baseline G1.

In general, the results produced by the dynamically-switched barrier demonstrate that the barrier switch at run time is not only possible within the G1 collector, but also provides improved throughput in certain scenarios, while imposing minimal additional overhead on top of static throughput-oriented barrier variants in terms of garbage collection pause times. At the same time, deficiencies in the current implementation of the dynamic switching limit its applicability in a broader context. A more sophisticated implementation of the dynamic barrier switch, leveraging different set of techniques to perform the barrier switch and minimize switch penalty, can be expected to offer more uniform benefits across wide set of applications, while preserving flexibility of selecting write barriers at run time.

Chapter 5

Conclusion

This chapter concludes the thesis, summarizes findings and provides an outline for possible future work in the area.

5.1 Summary

The main objective of this thesis has been the exploration of alternative throughput-oriented write barriers for the G1 collector and their impact on the throughput-pause time balance. This thesis proposes and implements two alternative write barriers, which along with necessary G1 adaptations in the stop-the-world pause constitute three throughput-oriented G1 variants. The alternative write barrier designs are based on eliminating parts of the existing G1 post-write barrier related to concurrent remembered set maintenance. Changes in garbage collection pause time accommodate these new write barriers. These new G1 variants have been evaluated on a comprehensive benchmark suite.

"Long" variant of the baseline G1 write barrier utilizes a conditional pre-filtering card marking barrier combined with partial concurrent refinement. Throughput gains produced by the "Long" variant are modest in most cases. With the exception for a large heap benchmark, garbage collection pause time impact of the "Long" barrier is limited, and pause time distributions exhibited widely intersect with the baseline G1 barrier, while median garbage collection pause times are slightly higher.

"Raw Parallel" variant of the G1 collector uses a minimal unconditional card marking barrier with no refinement. Throughput improvements produced by the "Raw Parallel" variant are larger than those of the "Long" variant, showing performance gains in a majority of benchmarks, in some cases approaching the performance of the Parallel Collector. At the same time, pause time costs are only marginally higher than that of the "Long" variant, with an exception for the same large heap benchmark, where pause times of this variant are longer than those of the baseline G1 and "Long" variant.

Dynamically-switched barrier allows run-time heuristics guided switching to and from the "Raw Parallel" variant. It shows a varying degree of throughput improvements, performing on par with the "Raw Parallel" variant in long-running benchmarks, but also degrading throughput in shorter ones. Specifically, the dynamic barrier switch is shown to improve application startup times. From a pause time perspective, the dynamically-switched barrier occupies a middle ground between the "Long" and "Raw Parallel" vari-

ants, also introducing own overhead. The reason is the simplistic and limited barrier switch mechanism.

Overall, the changes demonstrate significant throughput improvements at the expense of moderate increases in pause times for a substantial number of benchmarks. The cost of the throughput-oriented write barriers primarily depends on Java heap size — the larger the Java heap, the larger the impact on garbage collection pause times. While the proposed write barriers cannot fully replace the existing G1 barrier due to dependence on the heap size, this thesis shows that the throughput-oriented write barriers have a positive impact on G1 throughput-pause time balance for a significant number of benchmarks. From the perspective of the write barrier evaluation, this thesis also demonstrates that several widely known JVM benchmark suites are not well-suited for the write barrier impact characterization.

5.2 Future work

There are multiple avenues for future research on G1 collector throughput improvements via alternative write barriers. The barriers explored constitute only a fraction of possible write barrier implementations. These alternative barriers were directly derived from the existing G1 write barrier, ignoring other potential barrier candidates. Alternative write barrier designs that require more substantial changes in the G1 collector should be also interesting for investigation. Furthermore, even among the barrier and garbage collector adaptation options proposed in the thesis, only a few combinations were selected for thorough examination. Additional investigation on minimizing the pause time costs can significantly extend the number of cases when throughput-oriented write barriers are beneficial.

The dynamically-switched barrier proposed by this thesis uses a simplistic mechanism of barrier switching and a crude heuristic. Exploration of efficient, complete and less intrusive barrier switch approaches can lead to a considerable extension of dynamic barrier switching utility in various workloads.

In addition, an optimization of the G1 collector redirtying mechanism has been identified but put out of scope as it is independent of the write barrier. Nevertheless, research on G1 throughput improvement should not be limited with the write barriers, and extending the scope of changes to G1 can also be expected to bring significant throughput benefits, which would compound with improvements derived from the alternative write barriers.

Finally, despite the fact that the amount of benchmarking performed within the scope of this thesis is sufficient to demonstrate benefits and trade-offs offered by the throughput-oriented write barriers in G1, the write barriers devised in this thesis should be tested on a wider variety of workloads, garbage collector configurations (particularly, heap sizes), and hardware with different instruction set architectures to fully characterize the barrier behavior and more subtle implications in real-life scenarios.

Bibliography

[1] A. Adamson. "SPECjbb2005-A Year in the Life of a Benchmark". In: *2007 SPEC Benchmark Workshop*. 2007.

[2] B. Alpern et al. "The Jikes Research Virtual Machine Project: Building an Open-Source Research Community". In: *IBM Systems Journal* 44.2 (2005), pp. 399–417. ISSN: 0018-8670. DOI: 10.1147/sj.442.0399. URL: http://ieeexplore.ieee.org/document/5386722/ (visited on Feb. 28, 2023).

[3] Stephen M Blackburn. *Pjbb2005*. URL: https://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005/ (visited on Feb. 17, 2023).

[4] Stephen M Blackburn and Kathryn S. McKinley. "In or out?: Putting Write Barriers in Their Place". In: *Proceedings of the 3rd International Symposium on Memory Management*. ISMM02: 2002 International Symposium on Memory Management (Co-Located with PLDI 2002). Berlin Germany: ACM, June 20, 2002, pp. 175–184. ISBN: 978-1-58113-539-8. DOI: 10.1145/512429.512452. URL: https://dl.acm.org/doi/10.1145/512429.512452 (visited on Feb. 15, 2023).

[5] Stephen M. Blackburn and Antony L. Hosking. "Barriers: Friend or Foe?" In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM04: 2004 International Symposium on Memory Management (in Conjunction with OOPSLA 2004 Conference). Vancouver BC Canada: ACM, Oct. 24, 2004, pp. 143–151. ISBN: 978-1-58113-945-7. DOI: 10.1145/1029873.1029891. URL: https://dl.acm.org/doi/10.1145/1029873.1029891 (visited on Feb. 15, 2023).

[6] Stephen M. Blackburn et al. "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*. OOPSLA06: ACM SIGPLAN Object Oriented Programming Systems and Applications Conference. Portland Oregon USA: ACM, Oct. 16, 2006, pp. 169–190. ISBN: 978-1-59593-348-5. DOI: 10.1145/1167473.1167488. URL: https://dl.acm.org/doi/10.1145/1167473.1167488 (visited on Feb. 16, 2023).

[7] Man Cao. *JEP Draft: Throughput Post-Write Barrier for G1*. Sept. 5, 2019. URL: https://openjdk.org/jeps/8230187JEP%20draft:%20Throughput%20post-write%20barrier%20for%20G1 (visited on Feb. 28, 2023).

[8] Man Cao. *Reduce G1's CPU Cost with Simplified Write Post-Barrier and Disabling Concurrent Refinement*. JDK Bug System. Sept. 16, 2020. URL: https://bugs.openjdk.org/browse/JDK-8226197 (visited on Feb. 28, 2023).

[9] Stephen Cass. *Top Programming Languages 2022*. IEEE Spectrum. Aug. 23, 2022. URL: https://spectrum.ieee.org/top-programming-languages-2022 (visited on May 4, 2023).

[10] Oracle Corporation. *Java Platform, Standard Edition 20 Reference Implementations*. 2023. URL: https://jdk.java.net/java-se-ri/20 (visited on Feb. 28, 2023).

[11] Standard Performance Evaluation Corporation. *SPECjbb2005*. URL: https://www.spec.org/jbb2005/ (visited on Feb. 17, 2023).

[12] Standard Performance Evaluation Corporation. *SPECjvm® 2008*. URL: https://www.spec.org/jvm2008/ (visited on Feb. 17, 2023).

[13] Geoffrey De Smet. *How Much Faster Is Java 17?* URL: https://www.optaplanner.org/blog/2021/09/15/HowMuchFasterIsJava17.html (visited on Feb. 17, 2023).

[14] David Detlefs et al. "Garbage-First Garbage Collection". In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM04: 2004 International Symposium on Memory Management (in Conjunction with OOPSLA 2004 Conference). Vancouver BC Canada: ACM, Oct. 24, 2004, pp. 37–48. ISBN: 978-1-58113-945-7. DOI: 10.1145/1029873.1029879. URL: https://dl.acm.org/doi/10.1145/1029873.1029879 (visited on Feb. 15, 2023).

[15] David Dice. *False Sharing Induced by Card Table Marking*. Oracle Blogs. URL: https://web.archive.org/web/20170215224753/https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card (visited on Feb. 14, 2011).

[16] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. "Characterizing the Branch Misprediction Penalty". In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2006, pp. 48–58. ISBN: 1-4244-0186-0.

[17] Christine H. Flood and Roman Kennke. *JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)*. Aug. 28, 2021. URL: https://openjdk.org/jeps/189 (visited on Mar. 15, 2023).

[18] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*. OOPSLA07: ACM SIGPLAN Object Oriented Programming Systems and Applications Conference. Montreal Quebec Canada: ACM, Oct. 21, 2007, pp. 57–76. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. URL: https://dl.acm.org/doi/10.1145/1297027.1297033 (visited on Feb. 16, 2023).

[19] James Gosling et al. *The Java® Language Specification. Java SE 19 Edition*. Aug. 31, 2022. URL: https://docs.oracle.com/javase/specs/jls/se19/html/index.html (visited on Mar. 15, 2023).

[20] Webbug Group. *Dramatic Difference between UseConcMarkSweepGC and UseG1GC*. JDK Bug System. Feb. 11, 2019. URL: https://bugs.openjdk.org/browse/JDK-8062128 (visited on Feb. 17, 2023).

[21] Laurence Hellyer, Richard Jones, and Antony L. Hosking. "The Locality of Concurrent Write Barriers". In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10: International Symposium on Memory Management. Toronto Ontario Canada: ACM, June 5, 2010, pp. 83–92. ISBN: 978-1-4503-0054-4. DOI: 10.1145/1806651.1806666. URL: https://dl.acm.org/doi/10.1145/1806651.1806666 (visited on Feb. 15, 2023).

[22] Urs Hölzle. "A Fast Write Barrier for Generational Garbage Collectors". In: *OOPSLA/ECOOP*. Vol. 93. Citeseer, 1993.

[23] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. "A Comparative Performance Evaluation of Write Barrier Implementation". In: *ACM SIGPLAN Notices* 27.10 (Oct. 31, 1992), pp. 92–109. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/141937.141946. URL: https://dl.acm.org/doi/10.1145/141937.141946 (visited on Feb. 16, 2023).

[24] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide. Mar. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on Apr. 7, 2023).

[25] Stefan Johansson. *JEP 248: Make G1 the Default Garbage Collector*. Sept. 12, 2017. URL: https://openjdk.org/jeps/248 (visited on Feb. 28, 2023).

[26] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742: CRC Press, Aug. 19, 2011. ISBN: 978-1-4200-8279-1 978-1-315-38801-4. DOI: 10.1201/9781315388021. URL: https://www.taylorfrancis.com/books/9781315388014 (visited on Feb. 15, 2023).

[27] Roman Kennke. *JEP 304: Garbage Collector Interface*. Apr. 9, 2018. URL: https://openjdk.org/jeps/304 (visited on Mar. 15, 2023).

[28] B. Lang and F. Dupont. "Incremental Incrementally Compacting Garbage Collection". In: *ACM SIGPLAN Notices* 22.7 (July 1987), pp. 253–263. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/960114.29677. URL: https://dl.acm.org/doi/10.1145/960114.29677 (visited on May 5, 2023).

[29] Doug Lea. *The JSR-133 Cookbook for Compiler Writers*. Mar. 22, 2011. URL: https://gee.cs.oswego.edu/dl/jmm/cookbook.html (visited on Mar. 25, 2023).

[30] Philipp Lengauer et al. "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17: ACM/SPEC International Conference on Performance Engineering. L'Aquila Italy: ACM, Apr. 17, 2017, pp. 3–14. ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030211. URL: https://dl.acm.org/doi/10.1145/3030207.3030211 (visited on Feb. 17, 2023).

[31] Per Liden and Stefan Karlsson. *JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)*. Mar. 13, 2020. URL: https://openjdk.org/jeps/333 (visited on Mar. 10, 2023).

[32] Henry Lieberman and Carl Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". In: *Communications of the ACM* 26.6 (June 1983), pp. 419–429. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/358141.358147. URL: https://dl.acm.org/doi/10.1145/358141.358147 (visited on Mar. 23, 2023).

[33] Arm Limited. *Learn the Architecture - Memory Systems, Ordering, and Barriers*. June 10, 2022. URL: https://developer.arm.com/documentation/102336/0100 (visited on Apr. 10, 2023).

[34] Tim Lindholm et al. *The Java® Virtual Machine Specification. Java SE 19 Edition*. Aug. 31, 2022. URL: https://docs.oracle.com/javase/specs/jvms/se19/html/index.html (visited on Feb. 28, 2023).

[35] John McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367177.367199. URL: https://dl.acm.org/doi/10.1145/367177.367199 (visited on Mar. 16, 2023).

[36] OpenJDK. *Building the JDK*. URL: https://openjdk.org/groups/build/doc/building.html (visited on May 13, 2023).

[37] OpenJDK. *HotSpot Glossary of Terms*. URL: https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html (visited on Mar. 15, 2023).

[38] OpenJDK. *HotSpot Runtime Overview*. URL: https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html (visited on Mar. 15, 2023).

[39] OpenJDK. *Java Microbenchmark Harness (JMH)*. URL: https://github.com/openjdk/jmh (visited on May 12, 2023).

[40] OpenJDK. *JDK 20*. Jan. 24, 2023. URL: https://openjdk.org/projects/jdk/20/ (visited on Feb. 28, 2023).

[41] OpenJDK. *The HotSpot Group*. URL: https://openjdk.org/groups/hotspot/ (visited on Feb. 28, 2023).

[42] Oracle. *Available Collectors*. HotSpot Virtual Machine Garbage Collection Tuning Guide. URL: https://docs.oracle.com/en/java/javase/19/gctuning/available-collectors.html (visited on Mar. 15, 2023).

[43] Oracle. *cardTableBarrierSetAssembler_x86.Cpp*. JDK20. Aug. 30, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/cpu/x86/gc/shared/cardTableBarrierSetAssembler_x86.cpp (visited on Feb. 28, 2023).

[44] Oracle. *collectedHeap.Hpp*. JDK20. Nov. 3, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/share/gc/shared/collectedHeap.hpp (visited on Mar. 15, 2023).

[45] Oracle. *g1BarrierSetAssembler_x86.Cpp*. JDK20. Sept. 6, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/cpu/x86/gc/g1/g1BarrierSetAssembler_x86.cpp (visited on Feb. 28, 2023).

[46] Oracle. *g1BarrierSetC1.Cpp*. JDK20. Sept. 12, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/share/gc/g1/c1/g1BarrierSetC1.cpp (visited on Mar. 25, 2023).

[47] Oracle. *g1BarrierSetC2.Cpp*. JDK20. Oct. 26, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/share/gc/g1/c2/g1BarrierSetC2.cpp (visited on Mar. 25, 2023).

[48] Oracle. *g1DirtyCardQueue.Cpp*. JDK20. Nov. 25, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/share/gc/g1/g1DirtyCardQueue.cpp (visited on Mar. 25, 2023).

[49] Oracle. *g1YoungCollector.Cpp*. JDK20. Dec. 6, 2022. URL: https://github.com/openjdk/jdk20/blob/master/src/hotspot/share/gc/g1/g1YoungCollector.cpp (visited on Mar. 31, 2023).

[50] Oracle. *Garbage-First (G1) Garbage Collector*. HotSpot Virtual Machine Garbage Collection Tuning Guide. URL: https://docs.oracle.com/en/java/javase/19/gctuning/garbage-first-g1-garbage-collector1.html (visited on Feb. 28, 2023).

[51] Oracle. *Java Support for Large Memory Pages*. URL: https://www.oracle.com/java/technologies/javase/largememory-pages.html (visited on May 11, 2023).

[52] Oracle. *Java Virtual Machine Technology Overview*. Java Virtual Machine Guide. URL: https://docs.oracle.com/en/java/javase/19/vm/java-virtual-machine-technology-overview.html (visited on Mar. 15, 2023).

[53] Oracle. *The Java Command*. URL: https://docs.oracle.com/en/java/javase/20/docs/specs/man/java.html (visited on May 12, 2023).

[54] Oracle. *The Parallel Collector*. HotSpot Virtual Machine Garbage Collection Tuning Guide. URL: https://docs.oracle.com/en/java/javase/19/gctuning/parallel-collector1.html (visited on Mar. 1, 2023).

[55] Oracle. *The Z Garbage Collector*. HotSpot Virtual Machine Garbage Collection Tuning Guide. URL: https://docs.oracle.com/en/java/javase/19/gctuning/z-garbage-collector.html (visited on Mar. 10, 2023).

[56] Oracle. *Timeline of Key Java Milestones*. 2020. URL: https://www.oracle.com/java/moved-by-java/timeline/ (visited on Feb. 28, 2023).

[57] Oracle. *WriteBarrier.Java*. Version jdk20. Aug. 3, 2022. URL: https://github.com/openjdk/jdk20/blob/master/test/micro/org/openjdk/bench/vm/compiler/WriteBarrier.java (visited on Feb. 17, 2023).

[58] Aleksandar Prokopec et al. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '19: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. Phoenix AZ USA: ACM, June 8, 2019, pp. 31–47. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314637. URL: https://dl.acm.org/doi/10.1145/3314221.3314637 (visited on Feb. 16, 2023).

[59] Jevgenijs Protopopovs. *Jdk*. URL: https://github.com/protopopov1122/jdk.

[60] Jevgenijs Protopopovs. *Tp-Remset-Investigation-Benchmark-Runners*. URL: https://github.com/protopopov1122/tp-remset-investigation-benchmark-runners (visited on May 12, 2023).

[61] Thomas Schatzl. *Card Table Card Size Shenanigans*. Feb. 15, 2022. URL: https://tschatzl.github.io/2022/02/15/card-table-card-size.html (visited on June 19, 2023).

[62] Thomas Schatzl. *Concurrent Marking in G1*. Aug. 4, 2022. URL: https://tschatzl.github.io/2022/08/04/concurrent-marking.html (visited on Mar. 23, 2023).

[63] Thomas Schatzl. *Contention on Allocating New TLABs Constrains Throughput on G1*. JDK Bug System. July 16, 2019. URL: https://bugs.openjdk.org/browse/JDK-8131668 (visited on Mar. 13, 2023).

[64] Thomas Schatzl. *G1 20% Slower than Parallel in JRuby Rubykon Benchmark*. JDK Bug System. Nov. 15, 2022. URL: https://bugs.openjdk.org/browse/JDK-8253230 (visited on Feb. 17, 2023).

[65] Thomas Schatzl. *G1 Compares Badly to Parallel GC on Throughput on Javac Benchmark*. JDK Bug System. Sept. 16, 2020. URL: https://bugs.openjdk.org/browse/JDK-8132937 (visited on Feb. 17, 2023).

[66] Thomas Schatzl. *Guarantee Non-Humongous Object Allocation in Young Gen*. JDK Bug System. Apr. 6, 2022. URL: https://bugs.openjdk.org/browse/JDK-8191342 (visited on Mar. 13, 2023).

[67] Thomas Schatzl. *Purge CLDG Concurrently in G1*. JDK Bug System. Mar. 18, 2019. URL: https://bugs.openjdk.org/browse/JDK-8219643 (visited on Mar. 13, 2023).

[68] Thomas Schatzl. *Remove StoreLoad in G1 Post Barrier*. JDK Bug System. Sept. 16, 2020. URL: https://bugs.openjdk.org/browse/JDK-8226731 (visited on Mar. 13, 2023).

[69] Thomas Schatzl. *Threads May Do Significant Work out of the Non-Shared Overflow Buffer*. JDK Bug System. URL: https://bugs.openjdk.org/browse/JDK-8152438 (visited on Feb. 17, 2023).

[70] Aleksey Shipilev. *JEP 318: Epsilon: A No-Op Garbage Collector (Experimental)*. Sept. 24, 2018. URL: https://openjdk.org/jeps/318 (visited on Mar. 15, 2023).

[71] Patrick Sobalvarro. "A Lifetime-Based Garbage Collector for LISP Systems on General-Purpose Computers". In: (1988).

[72] TIOBE. *TIOBE Index for February 2023*. Feb. 2023. URL: https://www.tiobe.com/tiobe-index/ (visited on Feb. 28, 2023).

[73] P. R. Wilson and T. G. Moher. "A "Card-Marking" Scheme for Controlling Intergenerational References in Generation-Based Garbage Collection on Stock Hardware". In: *ACM SIGPLAN Notices* 24.5 (May 1989), pp. 87–92. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/66068.66077. URL: https://dl.acm.org/doi/10.1145/66068.66077 (visited on Feb. 15, 2023).

[74] Xi Yang et al. "Barriers Reconsidered, Friendlier Still!" In: *ACM SIGPLAN Notices* 47.11 (Jan. 8, 2013), pp. 37–48. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2426642.2259004. URL: https://dl.acm.org/doi/10.1145/2426642.2259004 (visited on Feb. 15, 2023).

[75] Taiichi Yuasa. "Real-Time Garbage Collection on General-Purpose Machines". In: *Journal of Systems and Software* 11.3 (1990), pp. 181–198.

[76] Wenyu Zhao and Stephen M. Blackburn. "Deconstructing the Garbage-First Collector". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Lausanne Switzerland: ACM, Mar. 17, 2020, pp. 15–29. ISBN: 978-1-4503-7554-2. DOI: 10.1145/3381052.3381320. URL: https://dl.acm.org/doi/10.1145/3381052.3381320 (visited on Feb. 15, 2023).

[77] Benjamin Zorn. *Barrier Methods for Garbage Collection*. Citeseer, 1990.

Appendix A

Source code listings

A.1 G1 post-write barrier for x86_64

```
; %r11 = store location
; %ebx = reference (compressed)
; %r12b = 0
;
; Fast-path
; Cross-region check:
da3b:  xorq    %r11, %r10
da3e:  shrq    $0x15, %r10
da42:  testq   %r10, %r10
da45:  je      0x22
; Null pointer check:
da47:  testl   %ebx, %ebx
da49:  je      0x1e
; Card address calculation:
da4b:  shrq    $0x9, %r11
da4f:  movabsq $0x7f48d0c00000, %rdi
da59:  addq    %r11, %rdi
da5c:  nopl    (%rax)
; Young gen. card check:
da60:  cmpb    $0x2, (%rdi)
da63:  jne     0x166 ; => slow-path

; Slow-path
; Thread-local dirty card buffer:
dbcf:  movq    0x48(%r15), %r10
dbd3:  movq    0x58(%r15), %r11
; Memory barrier:
dbd7:  lock
dbd8:  addl    $0x0, -0x40(%rsp)
dbdd:  nop
; Dirty card check:
dbe0:  cmpb    $0x0, (%rdi)
dbe3:  je      -0x180
; Dirty the card:
dbe9:  movb    %r12b, (%rdi)
; Check if the buffer is full
dbec:  testq   %r10, %r10
dbef:  jne     0x21
; If full => call runtime
```

```
dbf1:  movq    %r15, %rsi
dbf4:  nopl    (%rax,%rax)
dbfc:  nop
dc00:  movabsq $0x7f48f236f860, %r10
dc0a:  callq   *%r10
dc0d:  jmp     -0x1a9
; Else => enqueue the card
dc12:  movq    %rdi, -0x8(%r11,%r10)
dc17:  addq    $-0x8, %r10
dc1b:  movq    %r10, 0x48(%r15)
dc1f:  nop
dc20:  jmp     -0x1bc
```

A.2 "Raw Parallel" barrier for x86_64

```
; %r10 = store location
; %ebp = reference (compressed)
; %r12b = 0
;
c9211: shrq    $0x9, %r10
c9215: movabsq $0x7f4975093000, %r14
c921f: movb    %r12b, (%r14,%r10)
```

A.3 "Long" barrier for x86_64

```
; %r10 = store location
; %ebp = reference (compressed)
; %r12b = 0
;
; Cross-region check
c923d: xorq    %r11, %r10
c9240: shrq    $0x15, %r10
c9244: movabsq $0x7f0f5617b000, %r14
c924e: testq   %r10, %r10
c9251: je      0x1e
; Null pointer check
c9253: testl   %ebp, %ebp
c9255: je      0x1a
; Card address calculation
c9257: shrq    $0x9, %r11
c925b: movabsq $0x7f0f5617b000, %r10
c9265: addq    %r11, %r10
; Dirty chard check
c9268: cmpb    $0x0, (%r10)
c926c: je      0x3
; Dirty the card
c926e: movb    %r12b, (%r10)
```

Appendix B

Throughput-oriented garbage collector configuration

Throughput-oriented G1 garbage collector behavior is controlled by the JVM flags described in the table 4. These flags can be configured at startup time only in debug builds. In release builds their values need to be set at build time in accordance with the table 5 — the macros have to be specified in a form `-DTP_REMSET_INVESTIGATION_*_FLAG=true|false` in `extra-cxxflags` parameter of configuration script.

HotSpot flag	**Description**
`G1TpRemsetInvestigationRawParallelBarrier`	Controls whether the "Raw Parallel" barrier variant is enabled over the default "Long" barrier.
`G1TpRemsetInvestigationDirectUpdate`	Controls whether direct remembered set update optimization is applied.
`G1TpRemsetInvestigationPostevacRefine`	Controls whether post-evacuation refinement is enabled. The default behavior is disabled refinement.
`G1TpRemsetInvestigationConcurrentRefine`	Controls whether partial concurrent refinement is enabled; conflicts with the previous option.
`G1TpRemsetInvestigationDirtyChunkAtBarrier`	Controls whether chunk dirtying is included into the barrier.

Table 4: Throughput-oriented garbage collector configuration options

`G1TpRemsetInvestigation...`	**Macro name**
`RawParallelBarrier`	`RAW_PARALLEL_BARRIER`
`DirectUpdate`	`DIRECT_UPDATE`
`PostevacRefine`	`POSTEVAC_REFINE`
`ConcurrentRefine`	`CONCURRENT_REFINE`
`DirtyChunkAtBarrier`	`DIRTY_CHUNK_AT_BARRIER`

Table 5: Throughput-oriented garbage collector configuration macros

Therefore, the“Long” collector variant is configured as follows:

```
-DTP_REMSET_INVESTIGATION_RAW_PARALLEL_BARRIER_FLAG=false
-DTP_REMSET_INVESTIGATION_DIRECT_UPDATE_FLAG=false
-DTP_REMSET_INVESTIGATION_POSTEVAC_REFINE_FLAG=false
-DTP_REMSET_INVESTIGATION_CONCURRENT_REFINE_FLAG=true
-DTP_REMSET_INVESTIGATION_DIRTY_CHUNK_AT_BARRIER_FLAG=false
```

Whereas the options for “Raw Parallel” collector variant are:

```
-DTP_REMSET_INVESTIGATION_RAW_PARALLEL_BARRIER_FLAG=true
-DTP_REMSET_INVESTIGATION_DIRECT_UPDATE_FLAG=false
-DTP_REMSET_INVESTIGATION_POSTEVAC_REFINE_FLAG=false
-DTP_REMSET_INVESTIGATION_CONCURRENT_REFINE_FLAG=false
-DTP_REMSET_INVESTIGATION_DIRTY_CHUNK_AT_BARRIER_FLAG=false
```

Appendix C

Benchmark suite throughput results

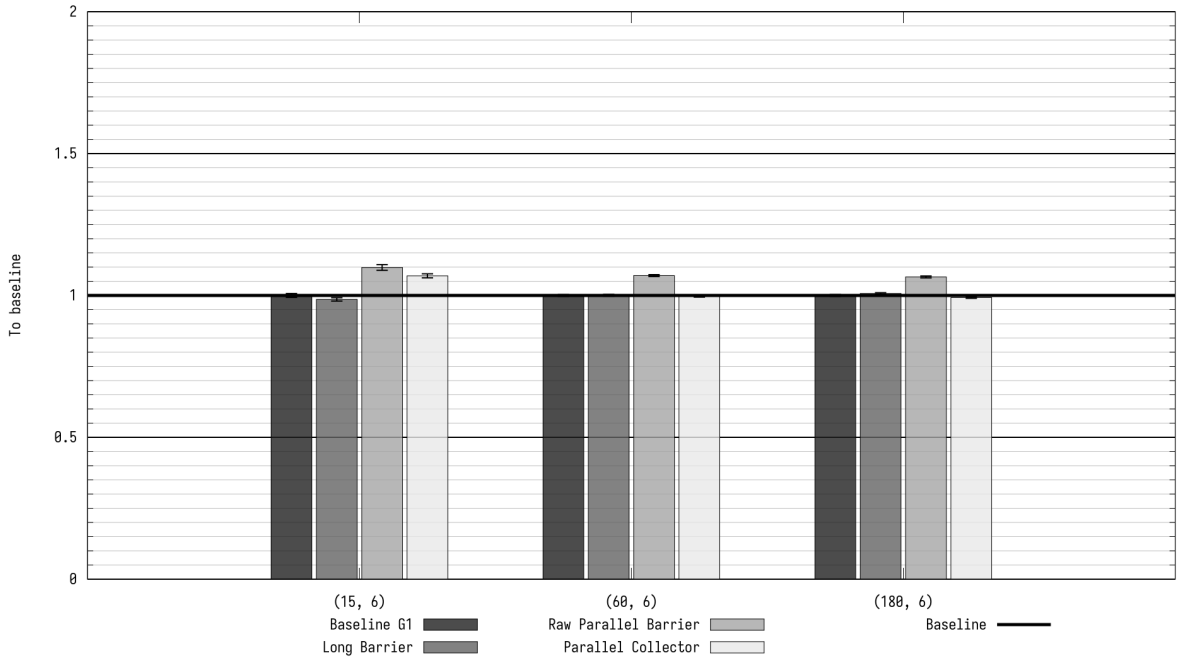


Figure 9: CompilerSpeed benchmark results

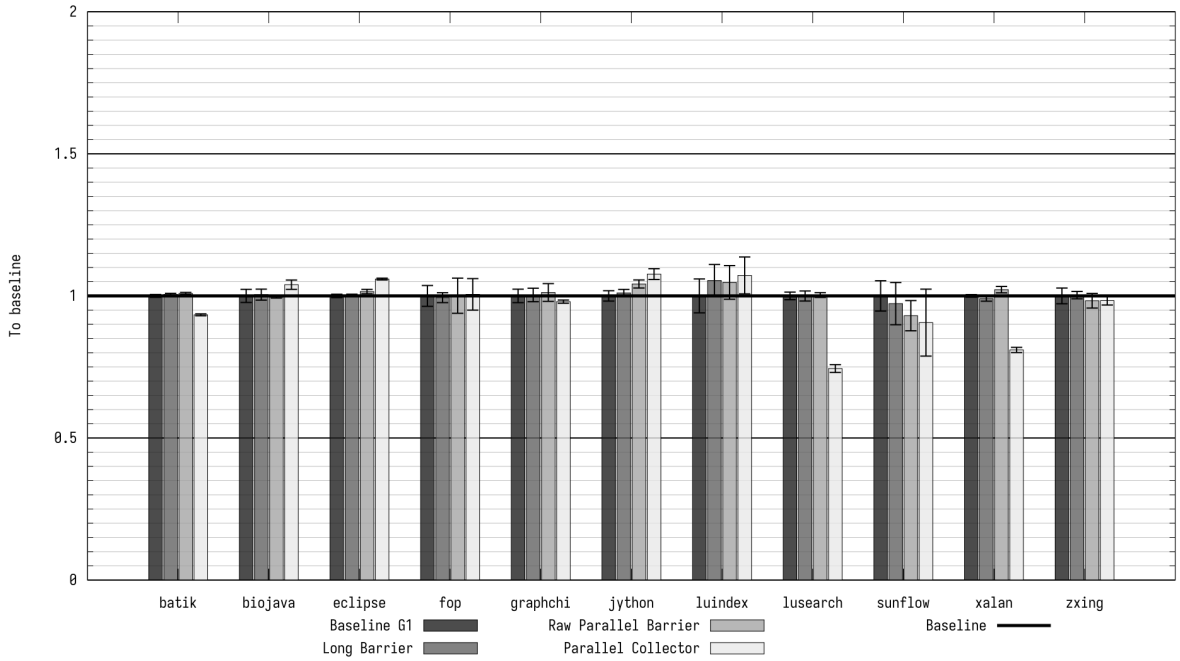


Figure 10: DaCapo benchmark results

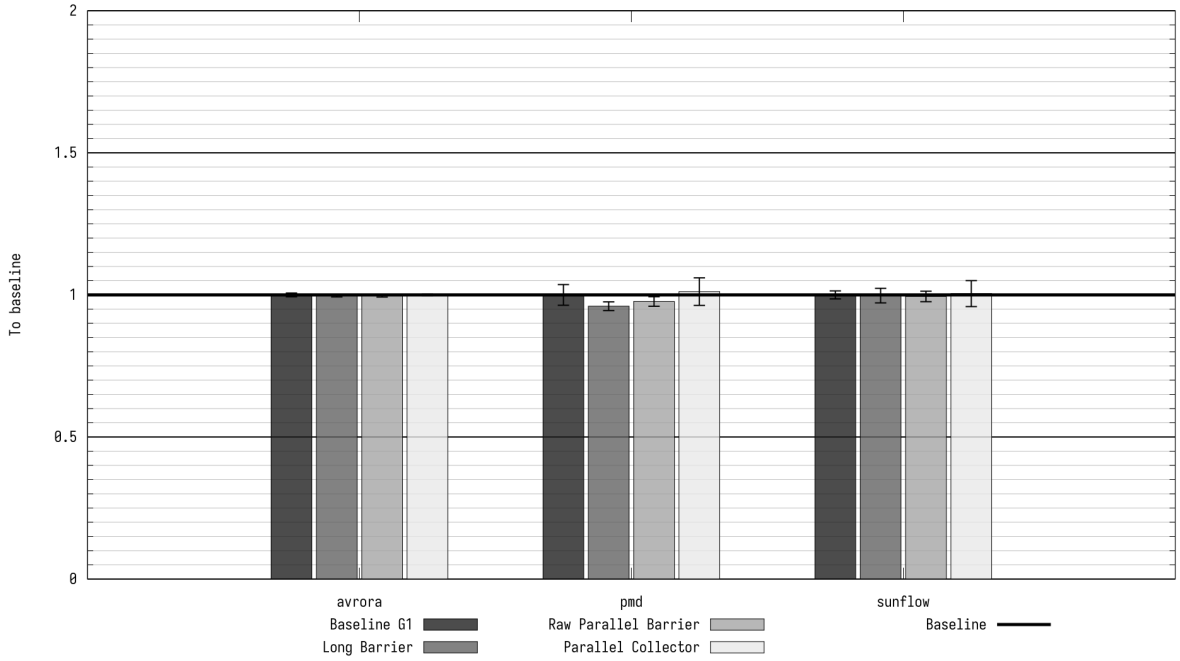


Figure 11: DaCapo large workload benchmark results

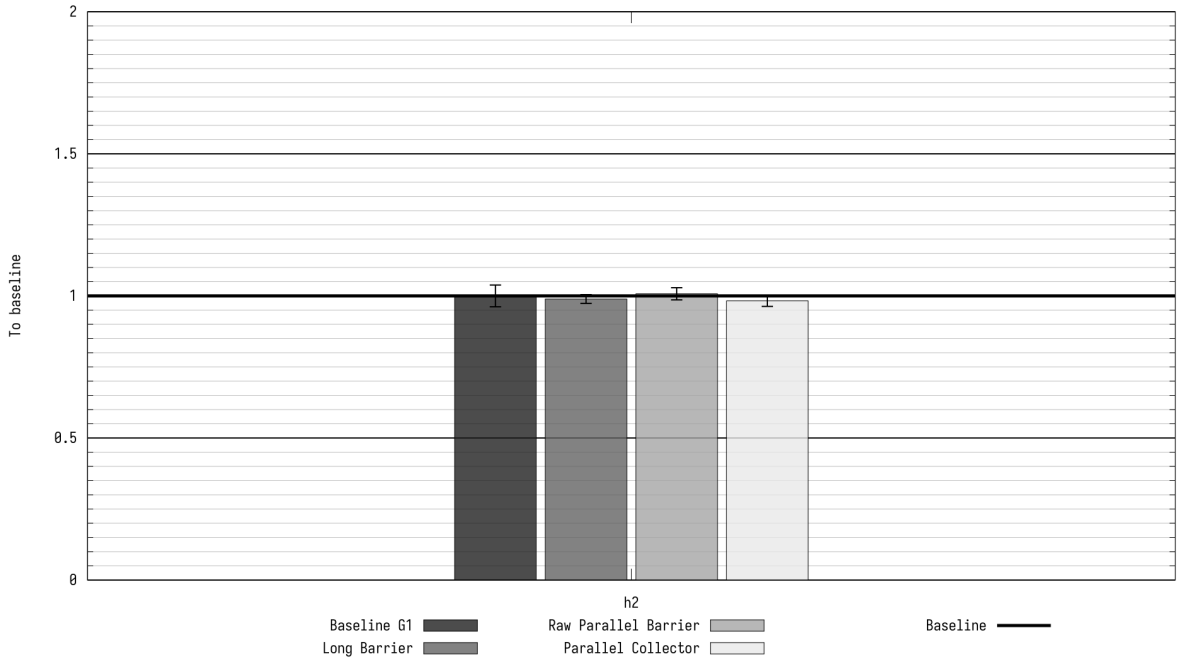


Figure 12: DaCapo huge workload benchmark results

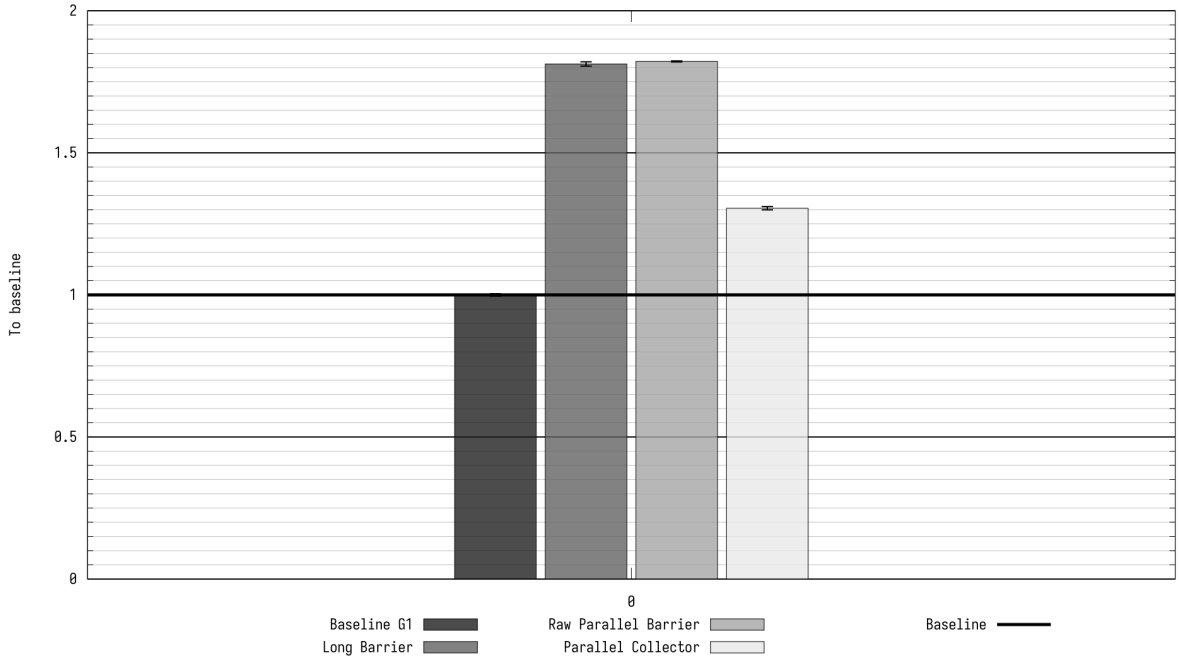


Figure 13: DelayInducer benchmark results

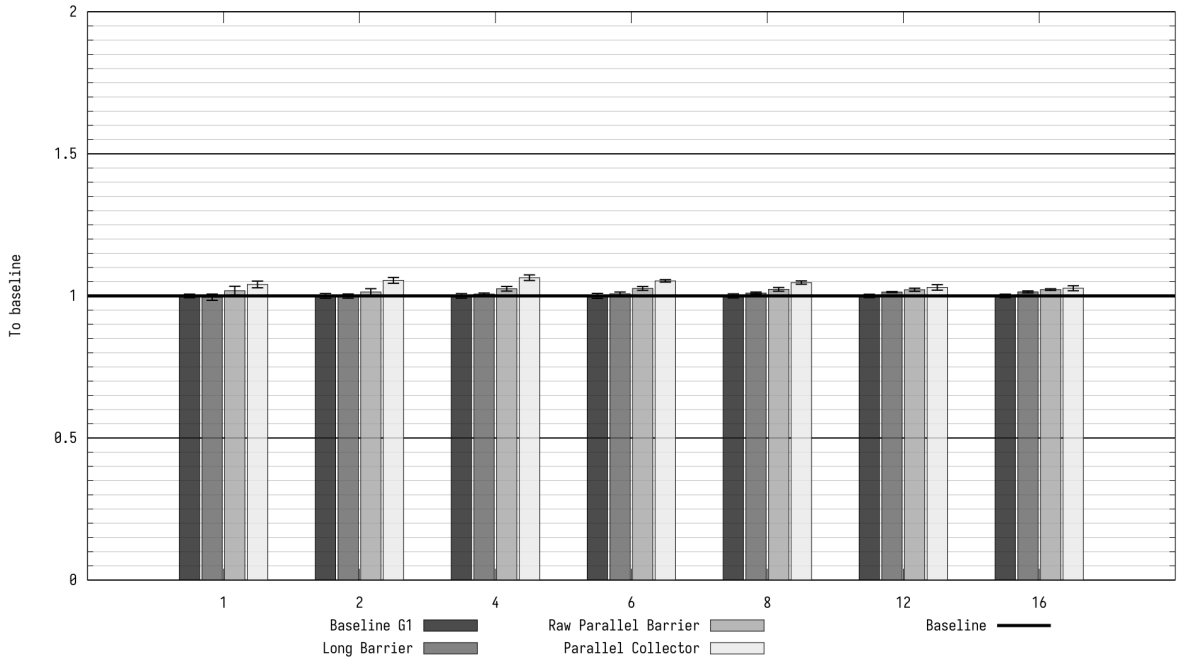


Figure 14: SPECjbb2005 benchmark results

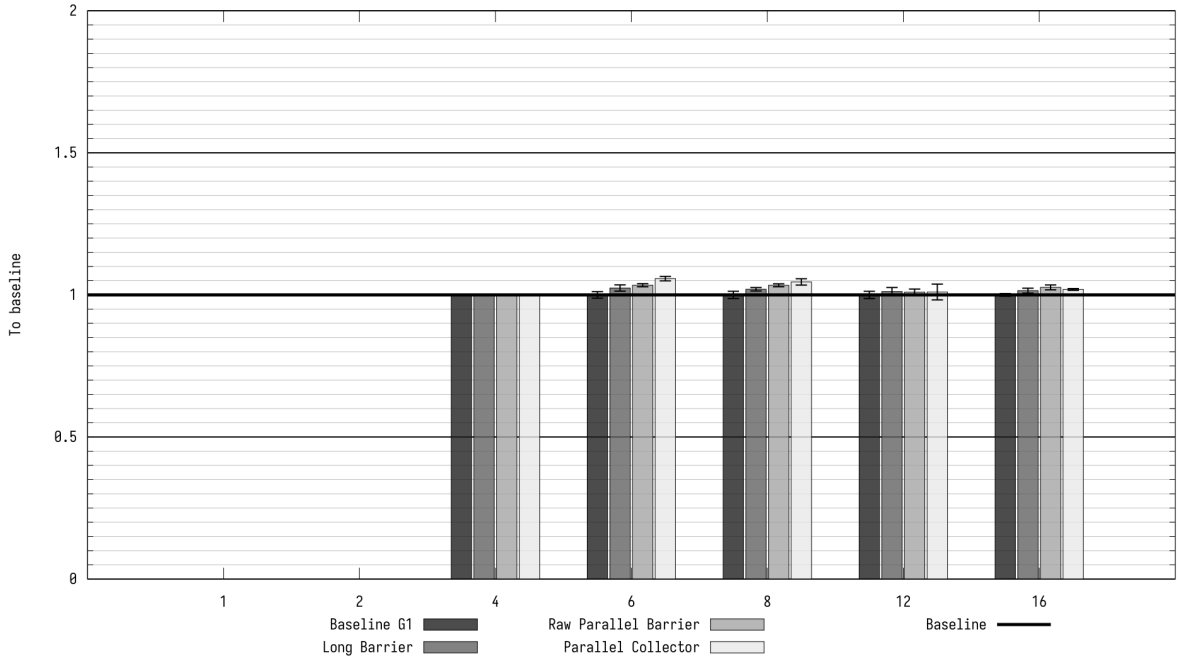


Figure 15: pjbb2005 benchmark results

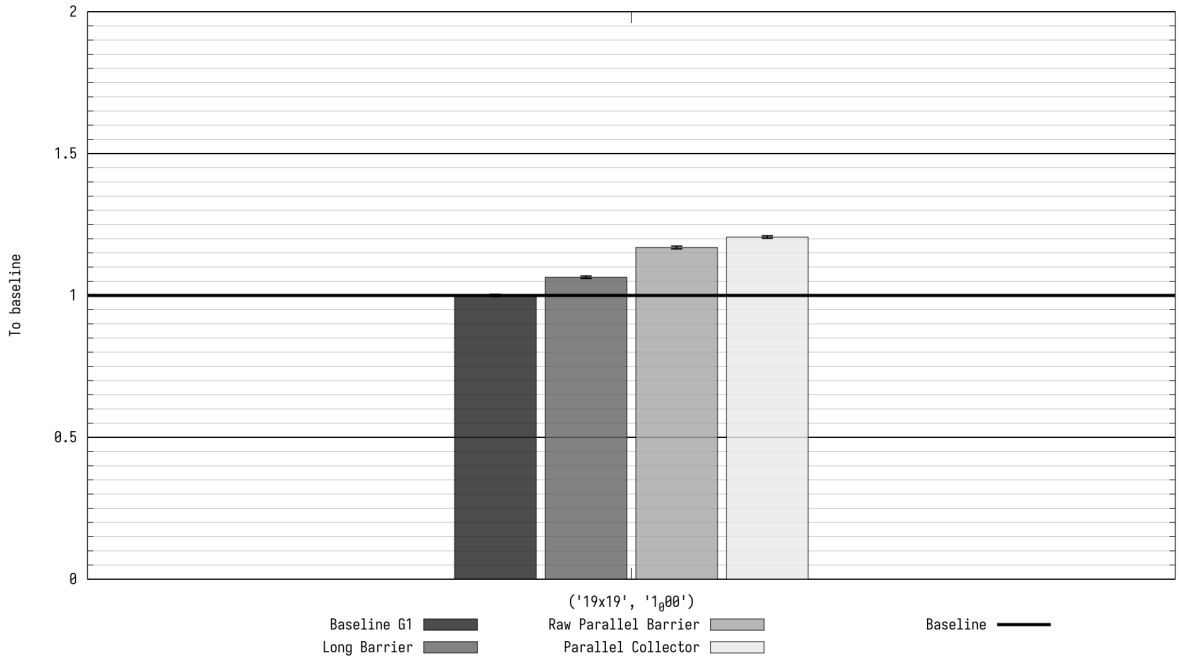


Figure 16: Rubykon benchmark results

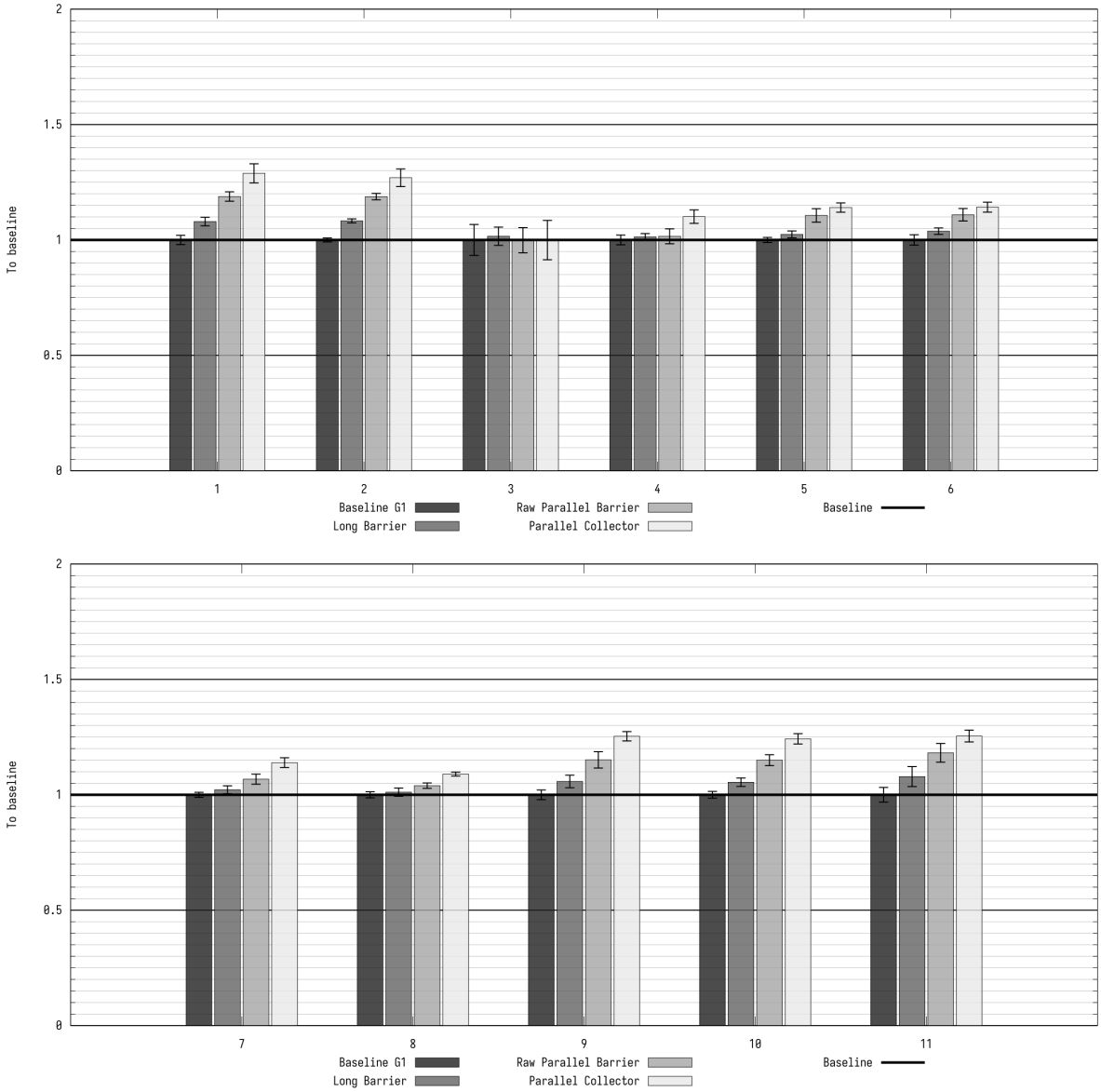


Figure 17: Optaplanner benchmark results

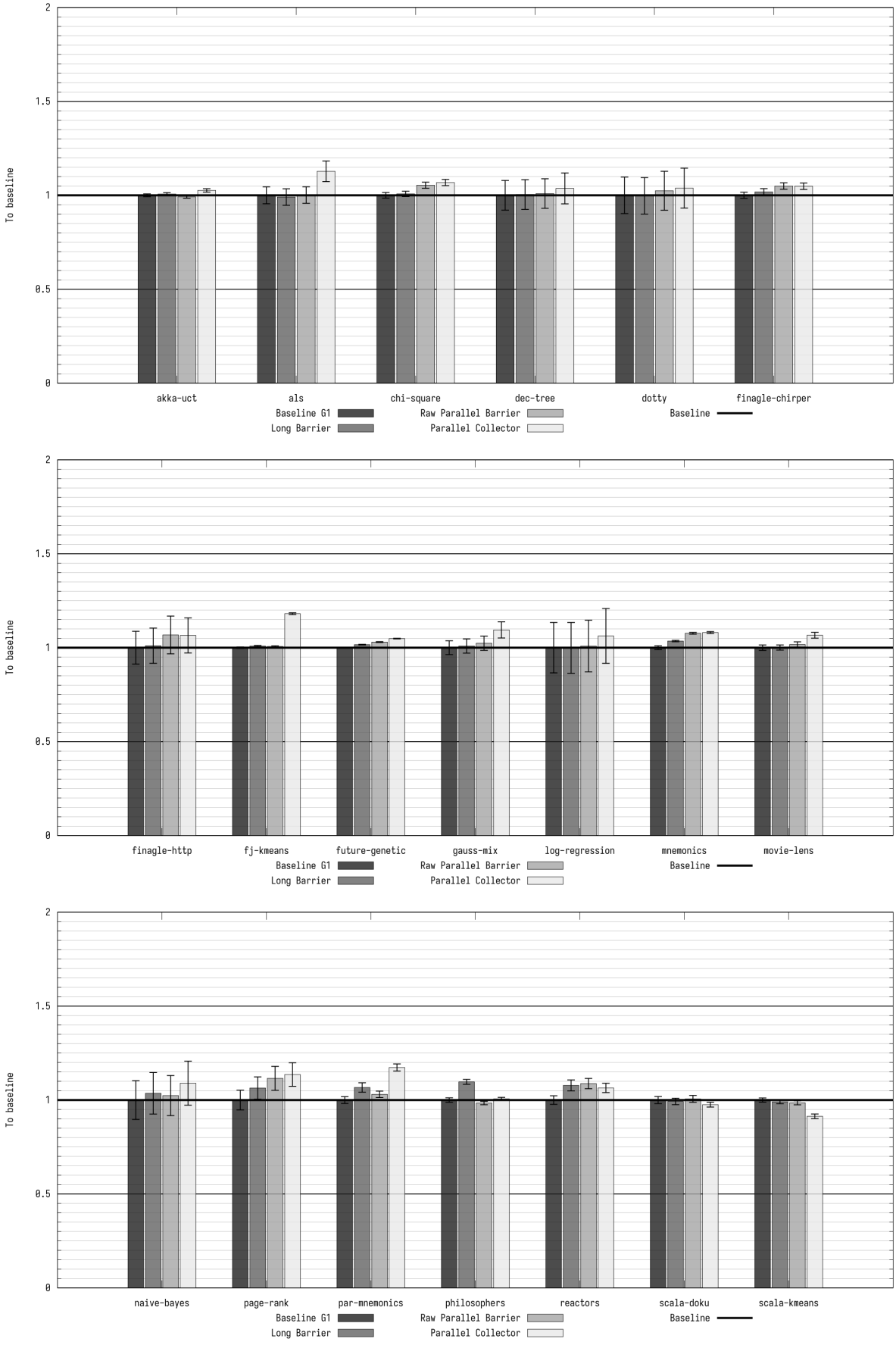


Figure 18: Renaissance benchmark results

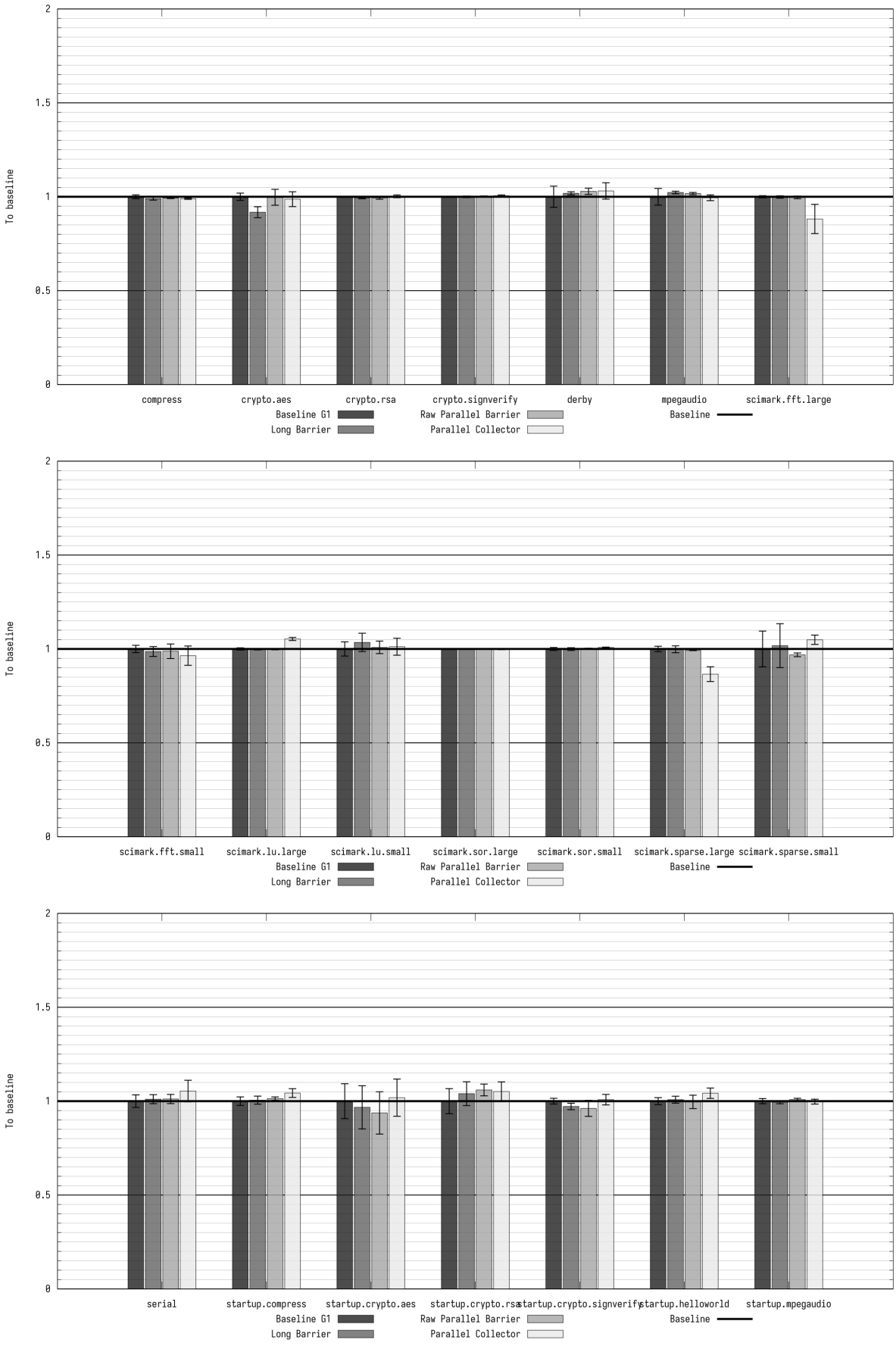


Figure 19: (Part 1) SPECjvm2008 benchmark results

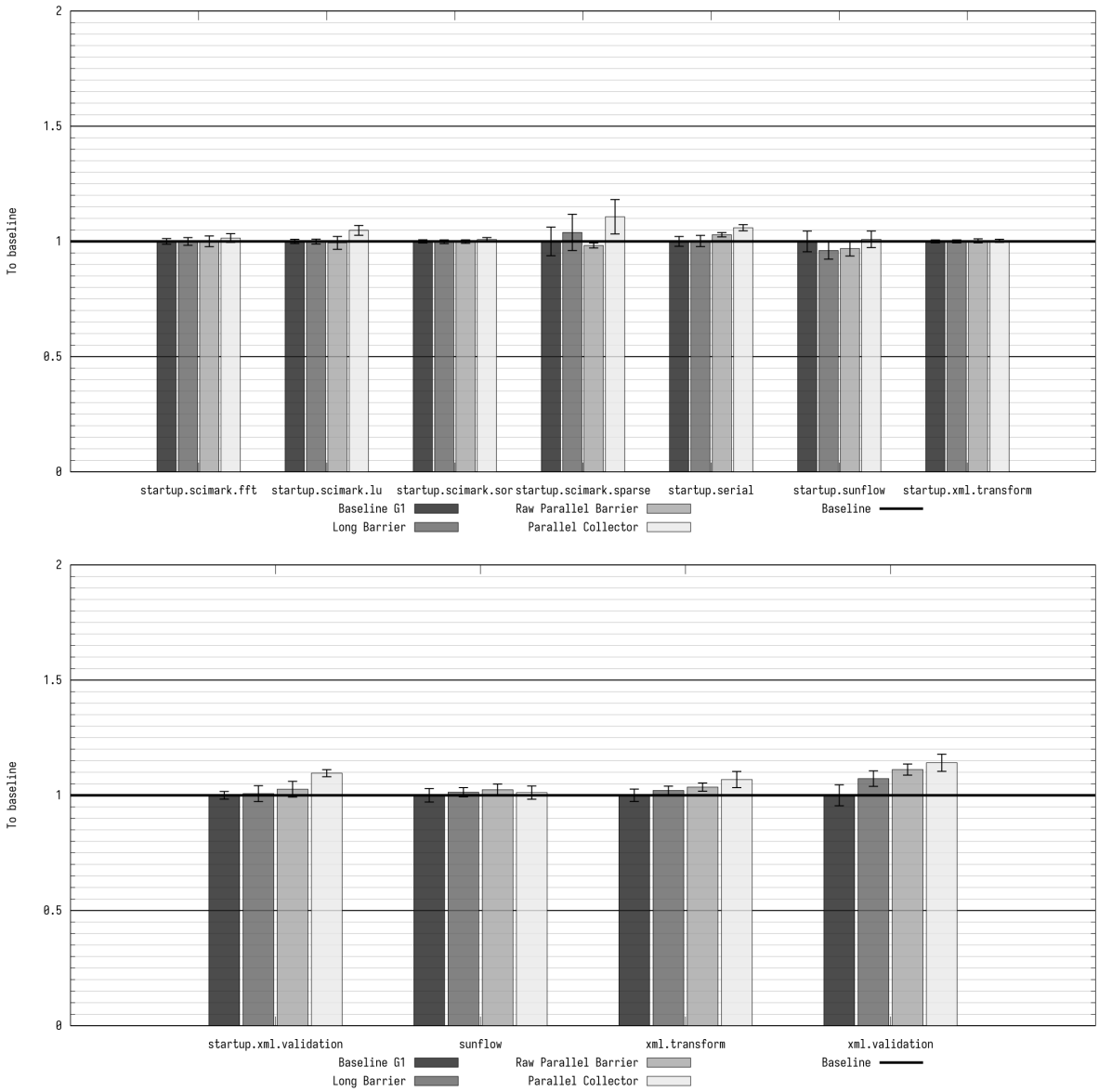


Figure 20: (Part 2) SPECjvm2008 benchmark results

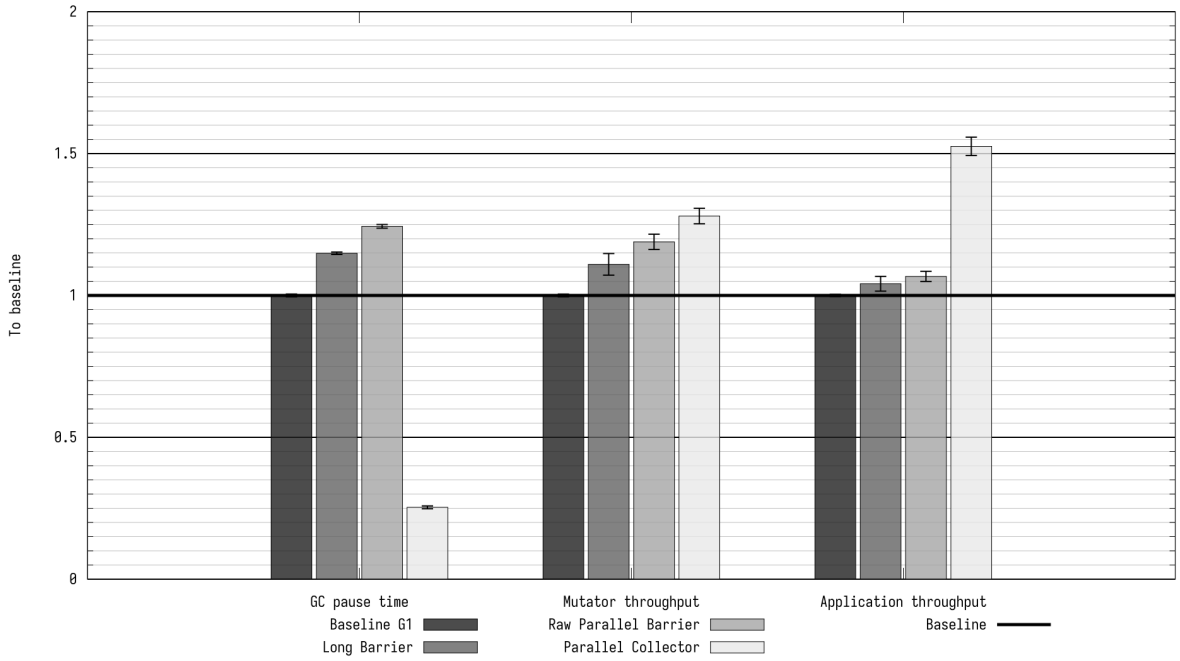


Figure 21: BigRamTester performance decomposition

Benchmark	Baseline G1	"Long" variant	"Raw Parallel" variant	Parallel Collector
CompilerSpeed	1.0000	0.9979	1.0778	1.0193
DaCapo	1.0000	0.9988	1.0020	0.9693
DelayInducer	1.0000	1.8127	1.8216	1.3050
SPECjbb2005	1.0000	1.0066	1.0215	1.0450
Optaplanner	1.0000	1.0431	1.1067	1.1712
pjbb2005	1.0000	1.0141	1.0207	1.0262
Renaissance	1.0000	1.0210	1.0291	1.0634
Rubykon	1.0000	1.0641	1.1691	1.2058
SPECjvm2008	1.0000	1.0022	1.0041	1.0180
BigRamTester	1.0000	1.0412	1.0672	1.5256

Table 6: Mean throughput of benchmark suite

Benchmark	Mean	5th Percentile	95th Percentile
CompilerSpeed			
15x6	1.0000	0.9931	1.0069
60x6	1.0000	0.9969	1.0031
180x6	1.0000	0.9968	1.0032
DaCapo			
batik	1.0000	0.9950	1.0050
biojava	1.0000	0.9769	1.0231
eclipse	1.0000	0.9941	1.0059
fop	1.0000	0.9634	1.0366
graphchi	1.0000	0.9763	1.0237
jython	1.0000	0.9818	1.0182
luindex	1.0000	0.9404	1.0596
lusearch	1.0000	0.9866	1.0134
sunflow	1.0000	0.9466	1.0534
xalan	1.0000	0.9954	1.0046
zxing	1.0000	0.9723	1.0277
avrora	1.0000	0.9934	1.0066
pmd	1.0000	0.9636	1.0364
sunflow (large)	1.0000	0.9859	1.0141
h2	1.0000	0.9617	1.0383
DelayInducer			
	1.0000	0.9959	1.0041
Optaplanner			
1	1.0000	0.9799	1.0201
2	1.0000	0.9914	1.0086
3	1.0000	0.9329	1.0671
4	1.0000	0.9789	1.0211
5	1.0000	0.9891	1.0109
6	1.0000	0.9775	1.0225
7	1.0000	0.9893	1.0107
8	1.0000	0.9868	1.0132
9	1.0000	0.9791	1.0209
10	1.0000	0.9854	1.0146
11	1.0000	0.9681	1.0319
pjbb2005			
1			
2			
4	1.0000	1.0000	1.0000

6	1.0000	0.9885	1.0115
8	1.0000	0.9873	1.0127
12	1.0000	0.9873	1.0127
16	1.0000	0.9953	1.0047
Renaissance			
akka-uct	1.0000	0.9924	1.0076
als	1.0000	0.9551	1.0449
chi-square	1.0000	0.9850	1.0150
dec-tree	1.0000	0.9210	1.0790
dotty	1.0000	0.9029	1.0971
finagle-chirper	1.0000	0.9836	1.0164
finagle-http	1.0000	0.9127	1.0873
fj-kmeans	1.0000	0.9962	1.0038
future-genetic	1.0000	0.9979	1.0021
gauss-mix	1.0000	0.9633	1.0367
log-regression	1.0000	0.8658	1.1342
mnemonics	1.0000	0.9897	1.0103
movie-lens	1.0000	0.9856	1.0144
naive-bayes	1.0000	0.8969	1.1031
page-rank	1.0000	0.9475	1.0525
par-mnemonics	1.0000	0.9819	1.0181
philosophers	1.0000	0.9882	1.0118
reactors	1.0000	0.9772	1.0228
scala-doku	1.0000	0.9811	1.0189
scala-kmeans	1.0000	0.9889	1.0111
Rubykon			
19x19 1000	1.0000	0.9956	1.0044
SPECjbb2005			
1	1.0000	0.9937	1.0063
2	1.0000	0.9918	1.0082
4	1.0000	0.9921	1.0079
6	1.0000	0.9913	1.0087
8	1.0000	0.9928	1.0072
12	1.0000	0.9941	1.0059
16	1.0000	0.9939	1.0061
SPECjvm2008			
compress	1.0000	0.9899	1.0101
crypto.aes	1.0000	0.9807	1.0193
crypto.rsa	1.0000	0.9979	1.0021
crypto.signverify	1.0000	0.9976	1.0024
derby	1.0000	0.9435	1.0565
mpegaudio	1.0000	0.9557	1.0443
scimark.fft.large	1.0000	0.9942	1.0058
scimark.fft.small	1.0000	0.9807	1.0193
scimark.lu.large	1.0000	0.9940	1.0060
scimark.lu.small	1.0000	0.9621	1.0379
scimark.sor.large	1.0000	0.9990	1.0010
scimark.sor.small	1.0000	0.9921	1.0079
scimark.sparse.large	1.0000	0.9856	1.0144
scimark.sparse.small	1.0000	0.9049	1.0951
serial	1.0000	0.9666	1.0334
startup.compress	1.0000	0.9774	1.0226
startup.crypto.aes	1.0000	0.9073	1.0927
startup.crypto.rsa	1.0000	0.9333	1.0667
startup.crypto.signverify	1.0000	0.9848	1.0152

startup.helloworld	1.0000	0.9813	1.0187
startup.mpegaudio	1.0000	0.9862	1.0138
startup.scimark.fft	1.0000	0.9877	1.0123
startup.scimark.lu	1.0000	0.9908	1.0092
startup.scimark.sor	1.0000	0.9930	1.0070
startup.scimark.sparse	1.0000	0.9378	1.0622
startup.serial	1.0000	0.9787	1.0213
startup.sunflow	1.0000	0.9550	1.0450
startup.xml.transform	1.0000	0.9936	1.0064
startup.xml.validation	1.0000	0.9834	1.0166
sunflow	1.0000	0.9708	1.0292
xml.transform	1.0000	0.9731	1.0269
xml.validation	1.0000	0.9545	1.0455

Table 7: Throughput of benchmark suite with the baseline G1

Benchmark	**Mean**	**5th Percentile**	**95th Percentile**
CompilerSpeed			
15x6	0.9861	0.9802	0.9920
60x6	1.0006	0.9973	1.0039
180x6	1.0071	1.0045	1.0097
DaCapo			
batik	1.0051	1.0011	1.0091
biojava	1.0044	0.9852	1.0238
eclipse	1.0034	1.0008	1.0060
fop	0.9937	0.9764	1.0112
graphchi	1.0033	0.9796	1.0271
jython	1.0105	0.9981	1.0229
luindex	1.0542	0.9980	1.1105
lusearch	0.9996	0.9819	1.0173
sunflow	0.9729	0.8985	1.0472
xalan	0.9905	0.9814	0.9996
zxing	1.0027	0.9897	1.0157
avrora	0.9968	0.9929	1.0007
pmd	0.9602	0.9449	0.9756
sunflow (large)	0.9975	0.9718	1.0232
h2	0.9890	0.9736	1.0045
DelayInducer			
	1.8126	1.8050	1.8201
Optaplanner			
1	1.0798	1.0612	1.0984
2	1.0827	1.0741	1.0914
3	1.0161	0.9765	1.0556
4	1.0132	0.9988	1.0276
5	1.0241	1.0092	1.0389
6	1.0380	1.0240	1.0520
7	1.0217	1.0046	1.0389
8	1.0114	0.9936	1.0292
9	1.0579	1.0305	1.0852
10	1.0544	1.0362	1.0726
11	1.0790	1.0357	1.1222
pjbb2005			
1			
2			

4	1.0000	1.0000	1.0000
6	1.0242	1.0132	1.0352
8	1.0198	1.0137	1.0259
12	1.0119	0.9977	1.0260
16	1.0149	1.0061	1.0236
Renaissance			
akka-uct	1.0067	0.9991	1.0142
als	0.9908	0.9468	1.0348
chi-square	1.0076	0.9933	1.0218
dec-tree	1.0040	0.9250	1.0830
dotty	0.9971	0.9000	1.0943
finagle-chirper	1.0179	1.0004	1.0355
finagle-http	1.0105	0.9168	1.1044
fj-kmeans	1.0088	1.0055	1.0121
future-genetic	1.0162	1.0143	1.0180
gauss-mix	1.0088	0.9710	1.0466
log-regression	0.9989	0.8636	1.1342
mnemonics	1.0354	1.0316	1.0392
movie-lens	1.0008	0.9871	1.0145
naive-bayes	1.0361	0.9255	1.1466
page-rank	1.0641	1.0048	1.1233
par-mnemonics	1.0669	1.0419	1.0919
philosophers	1.0967	1.0836	1.1098
reactors	1.0779	1.0488	1.1071
scala-doku	0.9926	0.9755	1.0095
scala-kmeans	0.9908	0.9806	1.0009
Rubykon			
19x19 1000	1.0641	1.0597	1.0685
SPECjbb2005			
1	0.9954	0.9844	1.0064
2	0.9994	0.9920	1.0068
4	1.0066	1.0031	1.0101
6	1.0073	1.0009	1.0137
8	1.0100	1.0063	1.0136
12	1.0135	1.0118	1.0153
16	1.0141	1.0105	1.0176
SPECjvm2008			
compress	0.9901	0.9820	0.9981
crypto.aes	0.9176	0.8884	0.9468
crypto.rsa	0.9928	0.9894	0.9962
crypto.signverify	0.9994	0.9953	1.0036
derby	1.0184	1.0102	1.0265
mpegaudio	1.0228	1.0159	1.0296
scimark.fft.large	0.9987	0.9922	1.0052
scimark.fft.small	0.9864	0.9601	1.0127
scimark.lu.large	0.9964	0.9946	0.9982
scimark.lu.small	1.0349	0.9861	1.0837
scimark.sor.large	0.9996	0.9982	1.0010
scimark.sor.small	0.9993	0.9920	1.0066
scimark.sparse.large	0.9983	0.9800	1.0167
scimark.sparse.small	1.0176	0.9008	1.1345
serial	1.0106	0.9866	1.0346
startup.compress	1.0053	0.9838	1.0268
startup.crypto.aes	0.9672	0.8522	1.0821
startup.crypto.rsa	1.0399	0.9765	1.1034

startup.crypto.signverify	0.9718	0.9546	0.9890
startup.helloworld	1.0078	0.9894	1.0262
startup.mpegaudio	0.9941	0.9863	1.0020
startup.scimark.fft	0.9998	0.9831	1.0165
startup.scimark.lu	0.9991	0.9884	1.0098
startup.scimark.sor	0.9984	0.9904	1.0065
startup.scimark.sparse	1.0390	0.9604	1.1175
startup.serial	1.0018	0.9776	1.0261
startup.sunflow	0.9606	0.9229	0.9982
startup.xml.transform	1.0000	0.9936	1.0064
startup.xml.validation	1.0073	0.9729	1.0417
sunflow	1.0133	0.9933	1.0333
xml.transform	1.0209	1.0021	1.0398
xml.validation	1.0724	1.0388	1.1061

Table 8: Throughput of benchmark suite with the "Long" variant

Benchmark	**Mean**	**5th Percentile**	**95th Percentile**
CompilerSpeed			
15x6	1.0985	1.0885	1.1086
60x6	1.0700	1.0666	1.0734
180x6	1.0652	1.0617	1.0686
DaCapo			
batik	1.0084	1.0046	1.0122
biojava	0.9956	0.9924	0.9988
eclipse	1.0152	1.0075	1.0231
fop	1.0007	0.9388	1.0626
graphchi	1.0120	0.9808	1.0433
jython	1.0420	1.0278	1.0562
luindex	1.0474	0.9883	1.1066
lusearch	1.0030	0.9947	1.0115
sunflow	0.9303	0.8772	0.9835
xalan	1.0220	1.0107	1.0333
zxing	0.9829	0.9572	1.0086
avrora	0.9973	0.9925	1.0020
pmd	0.9768	0.9601	0.9936
sunflow (large)	0.9945	0.9761	1.0130
h2	1.0075	0.9860	1.0289
DelayInducer			
	1.8215	1.8198	1.8235
Optaplanner			
1	1.1882	1.1679	1.2086
2	1.1879	1.1740	1.2017
3	0.9988	0.9441	1.0534
4	1.0157	0.9834	1.0480
5	1.1062	1.0769	1.1354
6	1.1092	1.0823	1.1362
7	1.0678	1.0459	1.0896
8	1.0396	1.0280	1.0512
9	1.1513	1.1159	1.1867
10	1.1501	1.1266	1.1736
11	1.1818	1.1412	1.2224
pjbb2005			
1			

2			
4	1.0000	1.0000	1.0000
6	1.0340	1.0286	1.0394
8	1.0338	1.0286	1.0390
12	1.0097	0.9988	1.0205
16	1.0264	1.0178	1.0350
Renaissance			
akka-uct	0.9917	0.9847	0.9987
als	1.0012	0.9573	1.0451
chi-square	1.0536	1.0373	1.0701
dec-tree	1.0094	0.9307	1.0879
dotty	1.0243	0.9207	1.1279
finagle-chirper	1.0494	1.0321	1.0666
finagle-http	1.0678	0.9674	1.1683
fj-kmeans	1.0066	1.0032	1.0100
future-genetic	1.0300	1.0276	1.0323
gauss-mix	1.0236	0.9857	1.0616
log-regression	1.0086	0.8709	1.1461
mnemonics	1.0767	1.0718	1.0815
movie-lens	1.0169	1.0026	1.0312
naive-bayes	1.0239	0.9171	1.1306
page-rank	1.1156	1.0517	1.1793
par-mnemonics	1.0305	1.0134	1.0476
philosophers	0.9841	0.9751	0.9929
reactors	1.0878	1.0605	1.1151
scala-doku	1.0062	0.9880	1.0245
scala-kmeans	0.9854	0.9748	0.9960
Rubykon			
19x19 1000	1.1691	1.1641	1.1741
SPECjbb2005			
1	1.0183	1.0027	1.0339
2	1.0137	1.0021	1.0254
4	1.0251	1.0169	1.0332
6	1.0262	1.0194	1.0330
8	1.0230	1.0165	1.0296
12	1.0216	1.0161	1.0272
16	1.0222	1.0193	1.0251
SPECjvm2008			
compress	0.9922	0.9903	0.9941
crypto.aes	0.9972	0.9548	1.0396
crypto.rsa	0.9937	0.9868	1.0005
crypto.signverify	1.0015	0.9985	1.0044
derby	1.0289	1.0124	1.0454
mpegaudio	1.0168	1.0108	1.0229
scimark.fft.large	0.9966	0.9897	1.0034
scimark.fft.small	0.9876	0.9488	1.0263
scimark.lu.large	0.9967	0.9954	0.9980
scimark.lu.small	1.0086	0.9752	1.0419
scimark.sor.large	1.0004	0.9995	1.0013
scimark.sor.small	1.0036	1.0032	1.0040
scimark.sparse.large	0.9918	0.9909	0.9927
scimark.sparse.small	0.9688	0.9588	0.9787
serial	1.0114	0.9870	1.0359
startup.compress	1.0135	1.0046	1.0225
startup.crypto.aes	0.9374	0.8247	1.0501

startup.crypto.rsa	1.0597	1.0283	1.0911
startup.crypto.signverify	0.9616	0.9190	1.0042
startup.helloworld	0.9961	0.9604	1.0317
startup.mpegaudio	1.0085	1.0016	1.0153
startup.scimark.fft	1.0004	0.9772	1.0237
startup.scimark.lu	0.9935	0.9656	1.0214
startup.scimark.sor	0.9991	0.9917	1.0066
startup.scimark.sparse	0.9823	0.9716	0.9930
startup.serial	1.0290	1.0197	1.0383
startup.sunflow	0.9691	0.9366	1.0017
startup.xml.transform	1.0025	0.9938	1.0111
startup.xml.validation	1.0264	0.9922	1.0606
sunflow	1.0241	0.9995	1.0488
xml.transform	1.0356	1.0178	1.0534
xml.validation	1.1117	1.0876	1.1358

Table 9: Throughput of benchmark suite with the "Raw Parallel" variant

Benchmark	**Mean**	**5th Percentile**	**95th Percentile**
CompilerSpeed			
15x6	1.0691	1.0620	1.0762
60x6	0.9982	0.9941	1.0022
180x6	0.9925	0.9899	0.9951
DaCapo			
batik	0.9334	0.9300	0.9368
biojava	1.0395	1.0231	1.0558
eclipse	1.0592	1.0561	1.0625
fop	1.0054	0.9499	1.0609
graphchi	0.9798	0.9742	0.9855
jython	1.0767	1.0576	1.0957
luindex	1.0720	1.0071	1.1370
lusearch	0.7443	0.7306	0.7581
sunflow	0.9060	0.7882	1.0238
xalan	0.8098	0.8007	0.8190
zxing	0.9834	0.9679	0.9989
avrora	1.0000	0.9975	1.0026
pmd	1.0116	0.9631	1.0600
sunflow (large)	1.0045	0.9588	1.0502
h2	0.9827	0.9632	1.0022
DelayInducer			
	1.3050	1.2992	1.3108
Optaplanner			
1	1.2886	1.2473	1.3300
2	1.2696	1.2315	1.3077
3	0.9994	0.9142	1.0846
4	1.1011	1.0720	1.1302
5	1.1403	1.1202	1.1605
6	1.1421	1.1204	1.1637
7	1.1394	1.1183	1.1605
8	1.0898	1.0811	1.0985
9	1.2536	1.2333	1.2739
10	1.2425	1.2201	1.2648
11	1.2546	1.2290	1.2801
pjbb2005			

1			
2			
4	1.0000	1.0000	1.0000
6	1.0573	1.0495	1.0651
8	1.0457	1.0346	1.0568
12	1.0102	0.9824	1.0380
16	1.0192	1.0161	1.0223
Renaissance			
akka-uct	1.0261	1.0169	1.0351
als	1.1278	1.0730	1.1827
chi-square	1.0680	1.0515	1.0846
dec-tree	1.0367	0.9545	1.1189
dotty	1.0382	0.9320	1.1444
finagle-chirper	1.0483	1.0313	1.0654
finagle-http	1.0653	0.9718	1.1588
fj-kmeans	1.1811	1.1763	1.1858
future-genetic	1.0483	1.0467	1.0501
gauss-mix	1.0946	1.0514	1.1376
log-regression	1.0626	0.9166	1.2085
mnemonics	1.0806	1.0756	1.0856
movie-lens	1.0661	1.0505	1.0817
naive-bayes	1.0899	0.9727	1.2073
page-rank	1.1358	1.0731	1.1985
par-mnemonics	1.1734	1.1544	1.1923
philosophers	1.0064	0.9985	1.0144
reactors	1.0646	1.0398	1.0894
scala-doku	0.9756	0.9628	0.9884
scala-kmeans	0.9135	0.9011	0.9258
Rubykon			
19x19 1000	1.2058	1.2014	1.2103
SPECjbb2005			
1	1.0403	1.0285	1.0521
2	1.0546	1.0442	1.0650
4	1.0641	1.0542	1.0739
6	1.0529	1.0486	1.0572
8	1.0469	1.0411	1.0528
12	1.0299	1.0202	1.0396
16	1.0270	1.0182	1.0357
SPECjvm2008			
compress	0.9897	0.9860	0.9934
crypto.aes	0.9870	0.9475	1.0266
crypto.rsa	1.0023	0.9949	1.0097
crypto.signverify	1.0055	1.0010	1.0099
derby	1.0310	0.9874	1.0746
mpegaudio	0.9946	0.9792	1.0100
scimark.fft.large	0.8817	0.8038	0.9595
scimark.fft.small	0.9642	0.9128	1.0157
scimark.lu.large	1.0532	1.0450	1.0614
scimark.lu.small	1.0119	0.9667	1.0570
scimark.sor.large	0.9975	0.9964	0.9987
scimark.sor.small	1.0080	1.0062	1.0099
scimark.sparse.large	0.8659	0.8265	0.9052
scimark.sparse.small	1.0490	1.0241	1.0739
serial	1.0540	0.9967	1.1114
startup.compress	1.0431	1.0198	1.0664

startup.crypto.aes	1.0186	0.9194	1.1179
startup.crypto.rsa	1.0511	0.9993	1.1028
startup.crypto.signverify	1.0079	0.9799	1.0359
startup.helloworld	1.0422	1.0146	1.0698
startup.mpegaudio	0.9974	0.9844	1.0105
startup.scimark.fft	1.0146	0.9955	1.0337
startup.scimark.lu	1.0478	1.0265	1.0691
startup.scimark.sor	1.0083	0.9999	1.0167
startup.scimark.sparse	1.1069	1.0322	1.1816
startup.serial	1.0594	1.0461	1.0727
startup.sunflow	1.0091	0.9731	1.0451
startup.xml.transform	1.0025	0.9956	1.0093
startup.xml.validation	1.0960	1.0806	1.1113
sunflow	1.0118	0.9830	1.0405
xml.transform	1.0684	1.0333	1.1035
xml.validation	1.1411	1.1039	1.1783

Table 10: Throughput of benchmark suite with the Parallel Collector

Benchmark	**Mean**	99% **Error**
Baseline G1		
ArrayFastPathNullLarge	1.0000	0.0322
ArrayFastPathNullSmall	1.0000	0.0382
ArrayFastPathRealLarge	1.0000	0.0016
ArrayFastPathRealSmall	1.0000	0.0290
FieldWriteBarrierFastPath	1.0000	0.0009
Geometric Mean	1.0000	
“Long” variant		
ArrayFastPathNullLarge	1.5783	0.0264
ArrayFastPathNullSmall	1.0407	0.0462
ArrayFastPathRealLarge	1.5785	0.0035
ArrayFastPathRealSmall	1.1901	0.0020
FieldWriteBarrierFastPath	1.2273	0.0011
Geometric Mean	1.3051	
“Raw Parallel” variant		
ArrayFastPathNullLarge	2.0092	0.0343
ArrayFastPathNullSmall	1.7612	0.0006
ArrayFastPathRealLarge	1.6215	0.0342
ArrayFastPathRealSmall	1.7120	0.0161
FieldWriteBarrierFastPath	1.2415	0.0011
Geometric Mean	1.6491	
Parallel Collector		
ArrayFastPathNullLarge	2.4231	0.0012
ArrayFastPathNullSmall	2.2619	0.0164
ArrayFastPathRealLarge	2.0859	0.0078
ArrayFastPathRealSmall	1.9316	0.0657
FieldWriteBarrierFastPath	1.7915	0.0019
Geometric Mean	2.0868	

Table 11: Results of WriteBarrier microbenchmark

Appendix D

Benchmark suite pause times

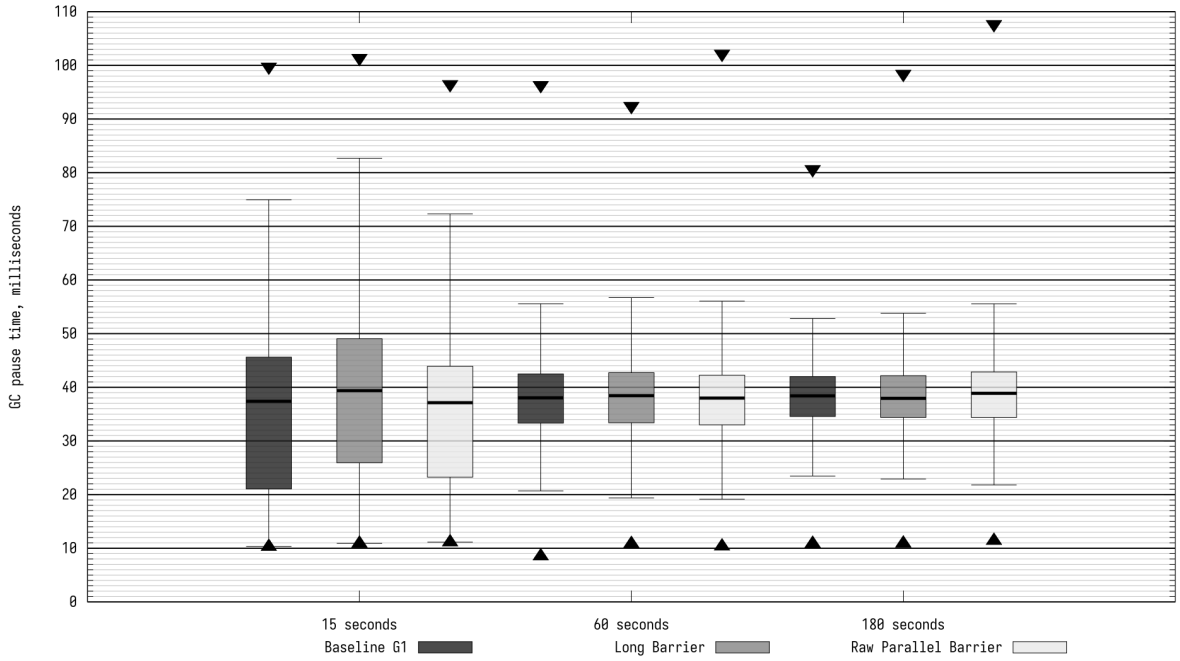


Figure 22: CompilerSpeed benchmark pause times

Variant	Mutator time	Pause time	Application run time
Absolute, milliseconds			
Baseline G1	3322393.18	17006.82	3339400
Long Barrier	3318196	18804	3337000
Raw Parallel Barrier	3318524.9	18075.1	3336600
Relative to application run time			
Baseline G1	0.9949	0.0051	1.0000
Long Barrier	0.9944	0.0056	1.0000
Raw Parallel Barrier	0.9946	0.0054	1.0000

Table 12: Optaplanner benchmark mutator and pause times

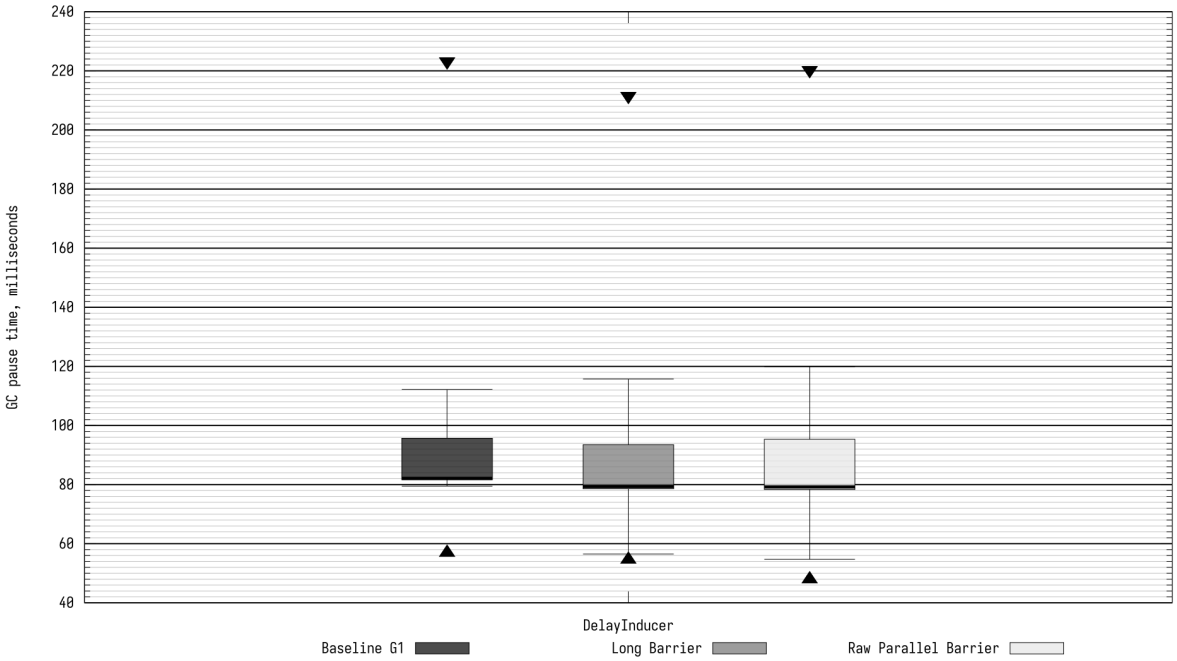


Figure 23: DelayInducer benchmark pause times

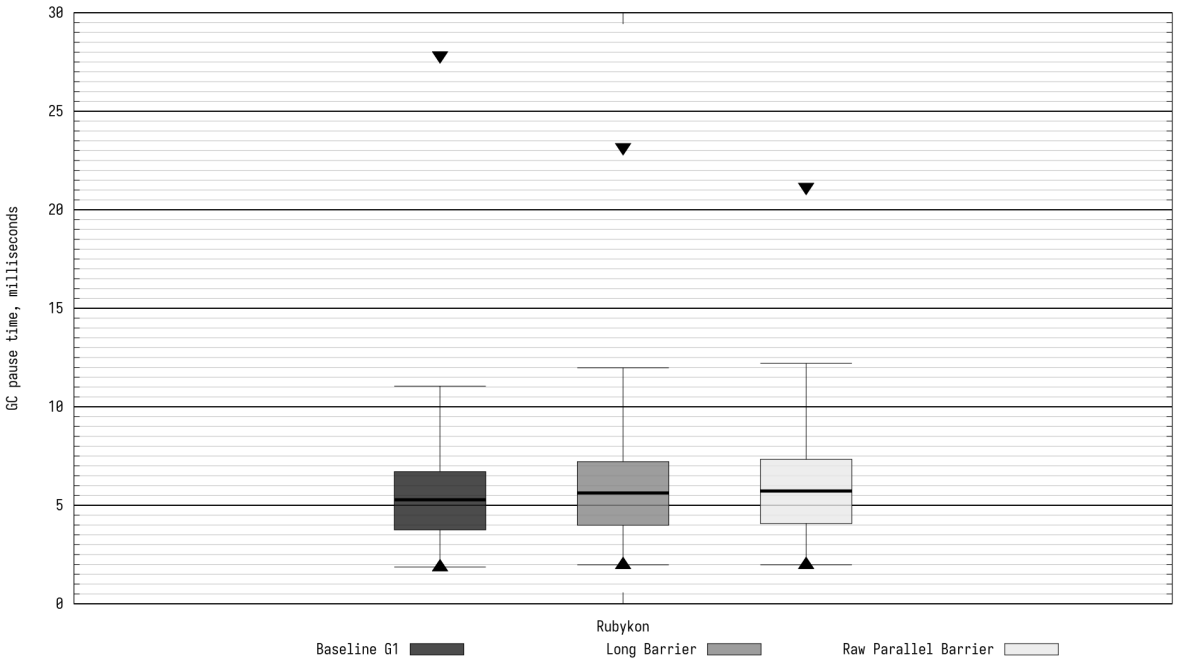


Figure 24: Rubykon benchmark pause times

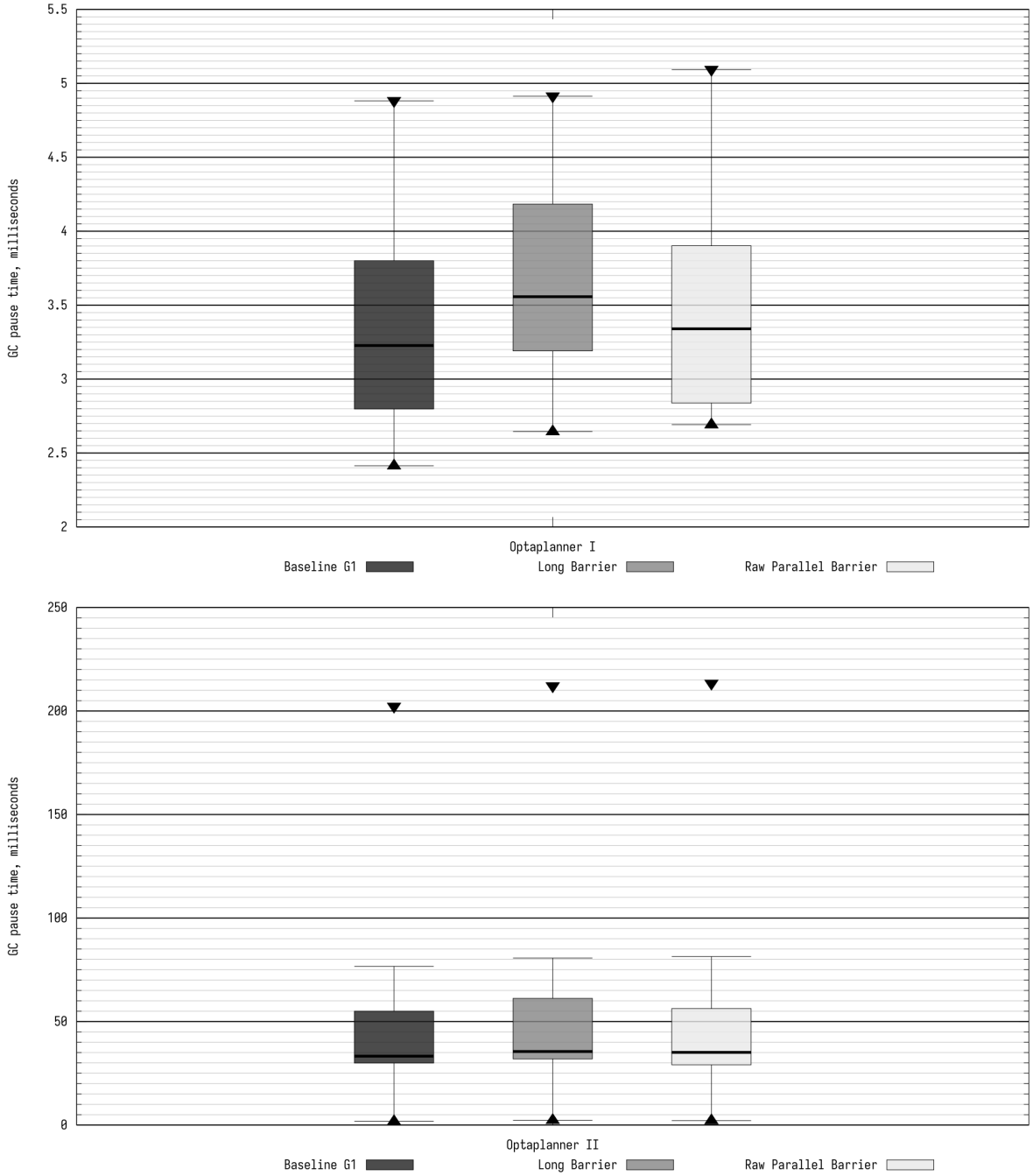


Figure 25: Optaplanner benchmark pause times

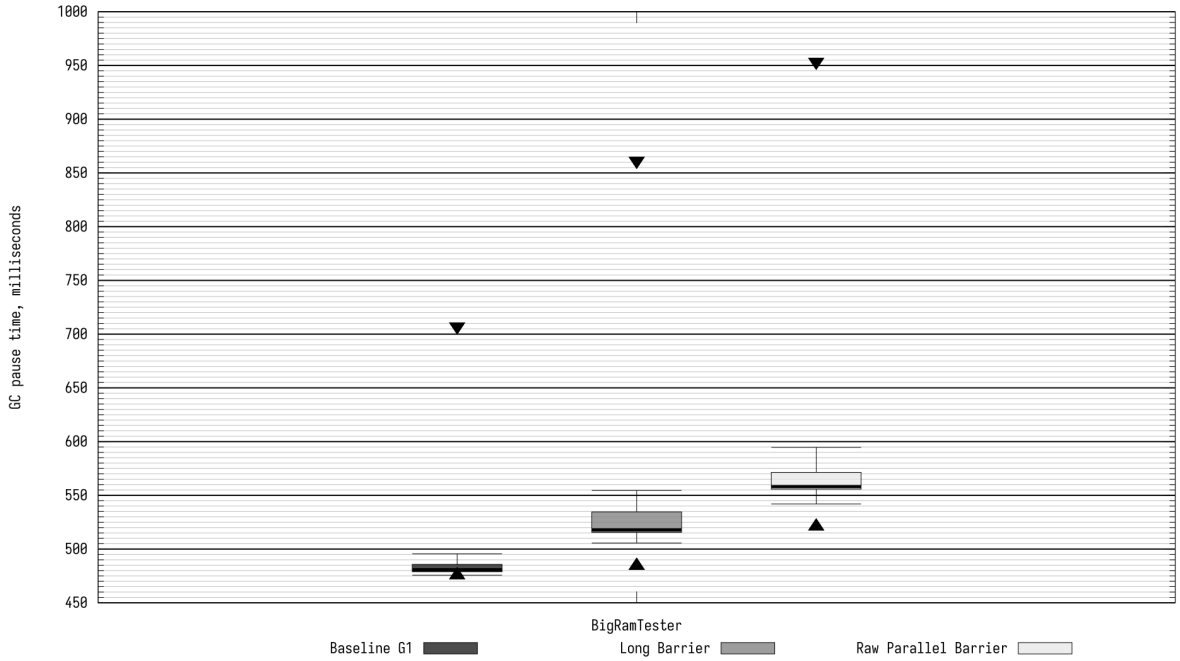


Figure 26: BigRamTester benchmark pause times

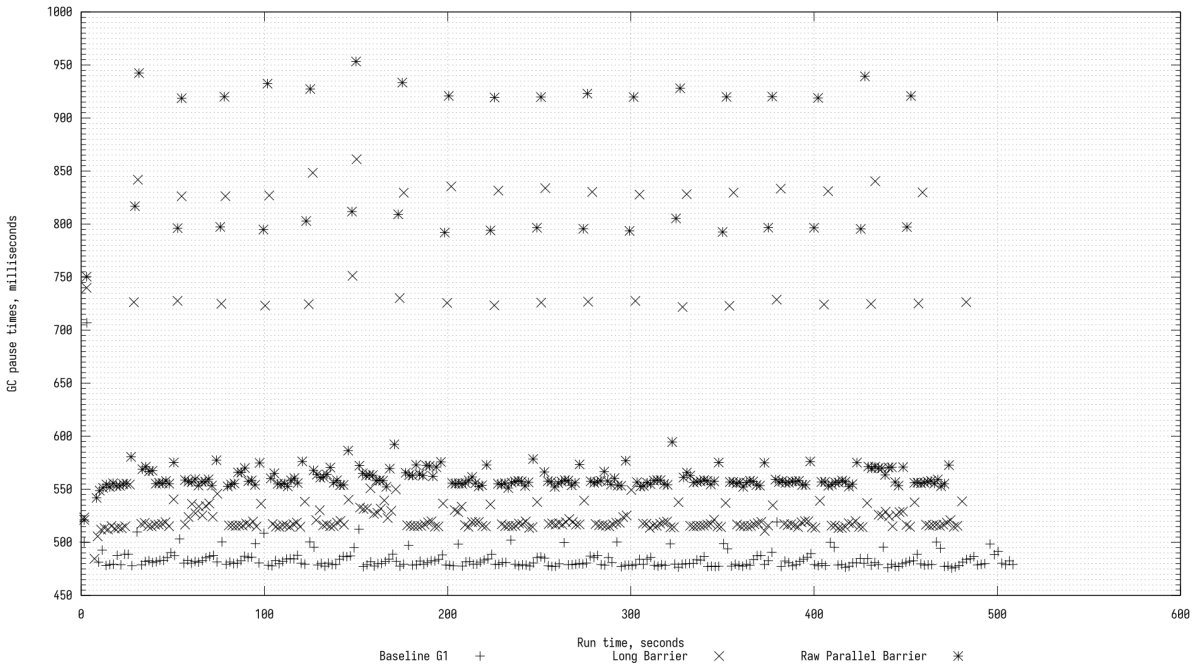


Figure 27: BigRamTester benchmark individual garbage collection pause times

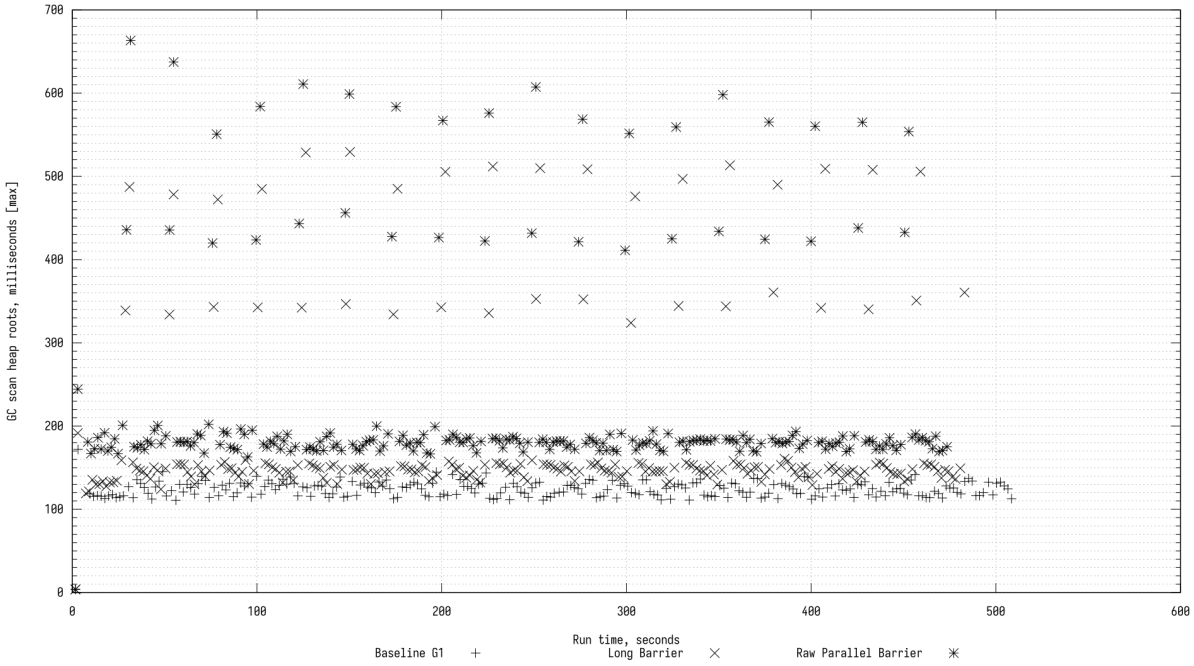


Figure 28: BigRamTester benchmark heap root scan phase durations

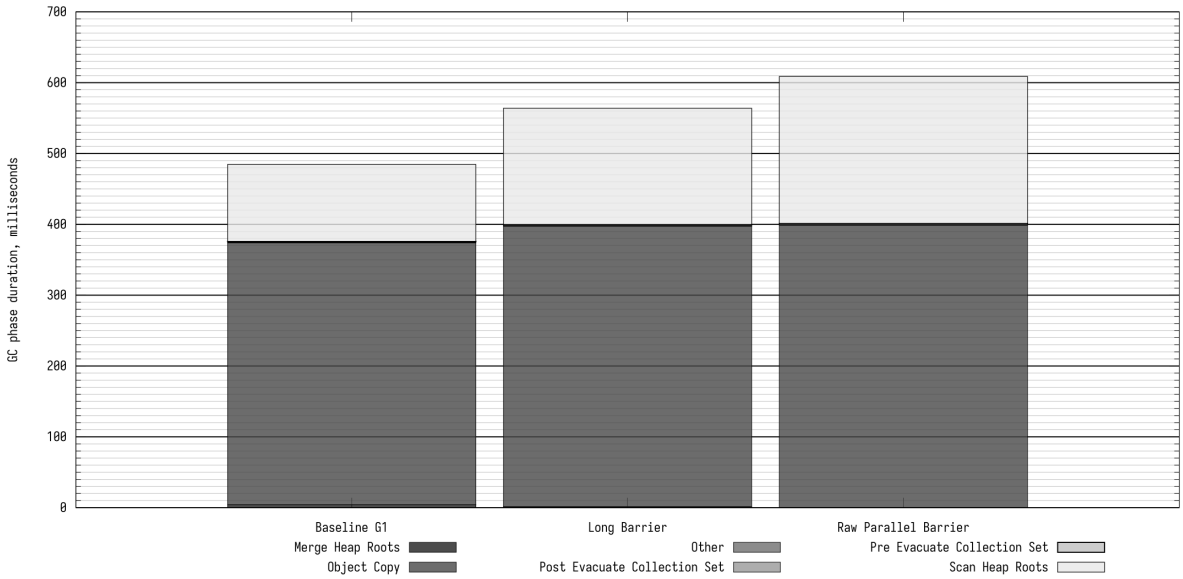


Figure 29: BigRamTester benchmark mean garbage collection phase durations

Benchmark	Mean	Median	-1.5 IQR	+1.5 IQR	Min	Max	Total
	Milliseconds						
DaCapo							
luindex	1.2226	1.11	0.865	1.381	0.865	3.242	62.353
zxing	1.7561	1.484	1.002	2.767	1.002	3.519	63.218
fop	4.6275	3.488	1.728	9.076	1.728	18.913	333.181
lusearch	1.0009	0.976	0.783	1.173	0.574	2.02	2309.176
batik	61.0769	43.4755	15.142	82.243	15.142	198.226	1710.152
eclipse	9.9093	5.951	1.453	27.622	1.453	34.429	2576.409
jython	2.2398	1.2155	0.811	2.101	0.811	13.01	1487.245
graphchi	1.9596	2.092	1.102	2.457	1.102	2.457	15.677
xalan	1.2194	1.194	1.048	1.348	0.889	11.668	6705.553
sunflow	1.7931	1.773	1.538	2.029	1.036	3.289	2960.409
biojava	3.5216	1.4875	1.144	3.766	1.144	29.141	647.972
pmd	7.7174	7.513	2.454	10.118	2.454	14.28	61.739
avrora	2.0553	1.709	1.096	4.269	1.096	4.269	59.603
sunflow (large)	2.2664	2.202	1.723	2.732	1.723	3.22	276.497
h2	82.6334	73.171	16.923	216.967	16.923	216.967	3718.501
Renaissance							
par-mnemonics	3.7157	3.0530	1.2520	7.3540	1.2520	26.9300	3641.4300
scala-doku	18.1136	19.3950	2.5650	31.1330	2.1350	33.4630	1430.9720
log-regression	3.5008	3.4070	2.1540	4.7360	2.1540	11.7580	889.1910
finagle-http	1.4123	1.3000	1.1110	1.5800	1.0810	6.5430	1687.7420
dec-tree	3.6340	2.8710	2.2410	8.7390	2.2410	9.8500	610.5140
gauss-mix	2.3021	2.1900	1.8570	2.5560	1.8150	5.1730	566.3280
future-genetic	2.1995	2.1570	1.5390	2.8080	1.4320	3.8060	2971.5460
fj-kmeans	2.8071	2.8790	1.8700	3.7260	1.7490	10.9790	3601.4810
finagle-chirper	4.4472	4.2650	3.1130	5.6100	3.1130	21.4390	12678.9580
scala-kmeans	6.9983	2.3950	1.1210	7.2860	1.1210	32.4790	48.9880
als	3.4513	2.9735	1.6550	5.6530	1.6550	30.7480	1822.2980
movie-lens	3.1802	2.9600	1.8010	4.5890	1.8010	8.0320	3377.3880
page-rank	126.1606	128.5890	3.2350	213.5390	3.2350	322.9280	19302.5670
naive-bayes	3.2357	3.0445	2.2950	3.7140	2.0910	9.8140	1216.6240
mnemonics	4.1293	2.8920	1.1220	6.4360	1.1220	95.4820	4872.5440
philosophers	2.0976	2.1400	0.7680	3.3980	0.7680	3.6030	10968.4880
reactors	49.7126	48.7095	1.3780	128.6280	1.3780	128.6280	5567.8150
chi-square	7.0031	5.0740	2.7080	16.2550	2.7080	18.9600	4699.0990
dotty	20.6530	21.2280	15.5960	25.4120	2.3670	31.5700	2499.0110
akka-uct	42.8692	43.2260	19.0420	67.1760	4.3740	77.4870	60831.3790
CompilerSpeed							
15 sec	35.6279	37.3840	10.3070	74.9520	10.3070	99.7190	24155.7070
60 sec	37.2280	38.0345	20.6860	55.5700	8.5430	96.2850	40652.9440
180 sec	37.6892	38.4110	23.4550	52.8210	10.8760	80.6020	60302.7550
Optaplanner							
Group I	3.3184	3.2270	2.4140	4.8820	2.4140	4.8820	225.6480
Group II	37.0009	33.2720	1.7560	76.6640	1.7560	202.1830	13505.3370
Other							
DelayInducer	100.3440	82.0950	79.4790	112.1590	56.9790	223.1250	80275.1730
pjbb2005	10.2362	10.3390	4.4880	17.2600	4.4880	33.5280	11597.5600
Rubykon	5.4944	5.2800	1.8640	11.0490	1.8640	27.8240	9488.8670

SPECjbb2005	11.5211	11.0610	6.0110	17.2020	3.5550	31.8280	8744.5200
SPECjvm2008	5.7196	3.3540	1.1610	6.6750	1.1610	52.2870	20047.0310
BigRamTester	484.2569	480.7530	475.7560	495.5820	475.7560	707.0220	117674.4260

1.5 IQR — bounds of $\pm 1.5 * IQR$ range; corresponds to the whiskers on pause time plots.

Table 13: Pause times of benchmark suite with Baseline G1

Benchmark	**Mean**	**Median**	**-1.5 IQR**	**+1.5 IQR**	**Min**	**Max**	**Total**
	Milliseconds						
DaCapo							
luindex	1.2654	1.16	0.927	1.436	0.927	3.266	68.332
zxing	1.7874	1.552	1.285	2.749	1.285	3.16	78.646
fop	4.8408	3.929	1.512	11.933	1.512	16.959	246.88
lusearch	1.0414	1.015	0.821	1.213	0.606	4.109	2003.662
batik	63.6793	45.358	9.955	93.968	9.955	195.541	1846.699
eclipse	12.1407	10.296	2.031	32.113	2.031	36.9	2391.716
jython	2.2918	1.31	0.822	2.491	0.822	13.564	1521.788
graphchi	2.2344	2.461	2.382	2.692	1.25	2.692	17.875
xalan	1.2448	1.222	1.09	1.358	0.927	11.511	6863.752
sunflow	1.8526	1.829	1.581	2.1	0.952	3.535	2039.76
biojava	3.7059	1.627	1.287	4.253	1.287	29.582	681.893
pmd	6.5545	6.5575	2.451	10.652	2.451	10.652	26.218
avrora	2.1661	1.765	0.939	4.214	0.939	4.214	62.816
sunflow (large)	2.3588	2.2655	1.934	2.825	1.934	3.277	240.598
h2	87.5947	83.542	17.227	223.789	17.227	223.789	3941.761
Renaissance							
par-mnemonics	3.7094	2.9110	1.2740	7.7450	1.2740	41.2220	3624.0360
scala-doku	19.5477	20.5255	2.7020	33.2080	2.7020	33.2080	1602.9100
log-regression	3.5706	3.4050	2.1220	5.1480	2.1220	11.3860	831.9590
finagle-http	1.4614	1.3500	1.1430	1.6330	1.1220	6.7920	1684.9410
dec-tree	3.8183	2.9940	2.5070	8.7330	2.5070	9.6780	656.7560
gauss-mix	2.3695	2.2485	1.9680	2.5540	1.9090	5.2510	582.9000
future-genetic	2.2221	2.1645	1.4370	2.9770	1.2570	5.7420	3004.2950
fj-kmeans	2.9079	2.9990	1.9010	3.7990	1.8750	10.5220	3722.0850
finagle-chirper	4.5279	4.3300	3.1940	5.7680	3.1940	16.5630	12664.4680
scala-kmeans	5.2729	3.0470	1.1090	5.3870	1.1090	20.5220	36.9100
als	3.6172	3.2310	1.9630	5.6620	1.9630	31.9630	1906.2650
movie-lens	3.3268	3.1320	1.8420	4.8830	1.8420	7.8160	3825.7820
page-rank	126.2425	117.0305	3.3520	216.7800	3.3520	325.9540	19946.3130
naive-bayes	3.4857	3.3155	2.4840	4.1050	2.3050	10.0820	1317.5920
mnemonics	4.5290	3.0400	1.0370	6.5010	1.0370	93.8640	5375.8810
philosophers	1.6785	1.6680	0.8010	2.6920	0.8010	3.2740	8001.4120
reactors	53.5849	48.9300	1.6270	137.9700	1.6270	137.9700	6215.8490
chi-square	7.1617	5.3350	3.2840	18.2110	3.2840	18.2110	4819.8170
dotty	21.8908	21.8395	14.2940	28.7330	2.6530	33.2580	2407.9830
akka-uct	44.5153	44.8035	20.0170	69.3470	4.6630	73.7160	62944.6930
CompilerSpeed							
15 sec	38.5834	39.3760	10.9060	82.6940	10.9060	101.3420	26236.6980
60 sec	37.8703	38.4390	19.3830	56.7050	10.8430	92.3840	40786.3110
180 sec	38.0583	37.9395	22.9090	53.7950	10.9230	98.3690	60436.5950

Optaplanner							
Group I	3.6713	3.5570	2.6450	4.9130	2.6450	4.9130	260.6620
Group II	38.7378	35.5150	2.2980	80.6580	2.2980	212.0350	14371.7190
Other							
DelayInducer	96.1223	79.3665	56.4890	115.7070	54.6640	211.4540	76897.8300
pjbb2005	10.6575	10.7495	4.7630	18.7650	4.7630	32.4700	12043.0060
Rubykon	5.8356	5.6240	1.9790	11.9790	1.9790	23.1650	10609.1740
SPECjbb2005	11.9452	11.3230	6.1790	17.0650	3.5410	31.5900	9197.8210
SPECjvm2008	6.0216	3.5715	1.3000	6.7260	1.3000	51.2710	21990.8660
BigRamTester	560.7872	517.8450	505.6840	554.5220	484.5990	861.1800	136271.2900

1.5 IQR — bounds of $\pm 1.5 * IQR$ range; corresponds to the whiskers on pause time plots.

Table 14: Pause times of benchmark suite with the "Long" variant

Benchmark	**Mean**	**Median**	**-1.5 IQR**	**+1.5 IQR**	**Min**	**Max**	**Total**
	Milliseconds						
DaCapo							
luindex	1.211	1.132	0.844	1.314	0.844	3.34	73.873
zxing	1.858	1.546	1.234	3.672	1.234	3.672	66.887
fop	4.7121	3.716	1.691	8.995	1.691	15.556	306.286
lusearch	1.0354	1.012	0.825	1.209	0.645	2.345	2001.409
batik	65.0242	45.1015	13.481	99.301	13.481	186.712	1820.678
eclipse	12.6327	10.034	1.912	34.456	1.912	34.456	2059.134
jython	2.2289	1.2045	0.782	2.208	0.782	13.361	1462.154
graphchi	2.2341	2.43	1.079	2.955	1.079	2.955	20.107
xalan	1.2499	1.227	1.109	1.359	0.941	11.795	6912.987
sunflow	1.8388	1.813	1.577	2.094	0.931	3.412	3035.817
biojava	4.0632	1.8805	1.381	3.494	1.381	29.958	747.631
pmd	6.593	7.4695	2.367	9.53	2.367	9.53	39.558
avrora	2.1131	1.758	1.133	4.487	1.133	4.487	61.281
sunflow (large)	2.3757	2.2985	1.986	2.807	1.986	3.677	242.326
h2	88.1958	82.506	17.618	217.921	17.618	221.538	3792.42
Renaissance							
par-mnemonics	3.8587	3.0580	1.2110	7.2070	1.2110	66.6450	3773.8470
scala-doku	19.3720	20.3770	2.5260	32.7870	2.5260	34.5640	1569.1320
log-regression	3.6526	3.4860	2.0190	5.1080	2.0190	11.4200	799.9130
finagle-http	1.4589	1.3430	1.1370	1.6270	1.1180	6.7680	1692.3130
dec-tree	3.8532	3.0940	2.5250	8.4480	2.5250	9.9730	662.7550
gauss-mix	2.3885	2.2510	1.9240	2.8130	1.9240	5.5920	587.5680
future-genetic	2.1979	2.1470	1.4640	2.9120	1.3970	3.7400	2969.3570
fj-kmeans	3.1183	3.2570	1.8790	4.3770	1.8790	12.2760	3975.7860
finagle-chirper	4.5423	4.3390	3.1460	5.7720	3.1460	9.4630	12514.1030
scala-kmeans	5.1801	2.4090	1.2000	5.5620	1.2000	20.5730	36.2610
als	3.6616	3.2460	1.9220	5.7680	1.9220	31.2760	1922.3650
movie-lens	3.3237	3.1220	1.8960	4.8380	1.8960	8.6620	3752.4310
page-rank	127.6821	141.4690	3.4970	326.3320	3.4970	326.3320	17237.0820
naive-bayes	3.5100	3.3625	2.3500	4.2890	2.2810	10.1200	1368.9060
mnemonics	4.5054	2.9000	0.9030	6.0660	0.9030	97.9070	5023.5540
philosophers	2.0000	2.0250	0.8170	3.5710	0.8170	4.4580	9062.2000
reactors	60.4364	68.0330	1.5210	154.6450	1.5210	154.6450	6587.5650

chi-square	7.2522	5.2640	3.0400	17.1280	3.0400	20.5470	4880.7210
dotty	22.6573	22.5185	17.3440	29.8180	2.9490	31.2850	2492.3030
akka-uct	45.6847	45.9685	20.7860	70.4380	5.2730	89.4190	64598.1340
CompilerSpeed							
15 sec	35.7892	37.1290	11.1490	72.3070	11.1490	96.4490	23799.8080
60 sec	37.4904	37.9875	19.1390	56.0530	10.3800	102.1120	41764.3250
180 sec	38.6305	38.8780	21.7900	55.5610	11.4280	107.6320	64165.1850
Optaplanner							
Group I	3.4427	3.3400	2.6920	5.0930	2.6920	5.0930	258.2040
Group II	38.7302	35.0955	2.1610	81.3860	2.1610	213.3050	14717.4820
Other							
DelayInducer	96.8066	79.1930	54.7080	119.9020	48.0030	220.1870	77445.2420
pjbb2005	10.6529	10.7870	4.7510	17.5870	4.7510	32.5130	12037.8050
Rubykon	5.8830	5.7255	1.9810	12.2050	1.9810	21.1490	11613.1180
SPECjbb2005	12.0602	12.0375	6.2480	17.4370	3.6270	32.5840	9286.3700
SPECjvm2008	6.1772	3.6130	1.2200	6.8850	1.2200	49.0990	22905.0120
BigRamTester	605.7378	558.1280	541.9700	594.6250	521.1840	953.3130	147194.2820

1.5 IQR — bounds of $\pm 1.5 * IQR$ range; corresponds to the whiskers on pause time plots.

Table 15: Pause times of benchmark suite with the "Raw Parallel" variant

Phase	**Duration, milliseconds**
Baseline G1	
Merge Heap Roots	4.1609
Object Copy	369.5757
Other	0.7330
Post Evacuate Collection Set	1.0494
Pre Evacuate Collection Set	0.2070
Scan Heap Roots	108.5309
"Long" variant	
Merge Heap Roots	1.4160
Object Copy	394.1407
Other	0.7963
Post Evacuate Collection Set	1.3053
Pre Evacuate Collection Set	0.2033
Scan Heap Roots	162.9255
"Raw Parallel" variant	
Merge Heap Roots	0.4391
Object Copy	396.4667
Other	0.7826
Post Evacuate Collection Set	1.6165
Pre Evacuate Collection Set	0.2041
Scan Heap Roots	206.2288

Table 16: BigRamTester benchmark garbage collection phase durations

Appendix E

Chunk table modification benchmark results

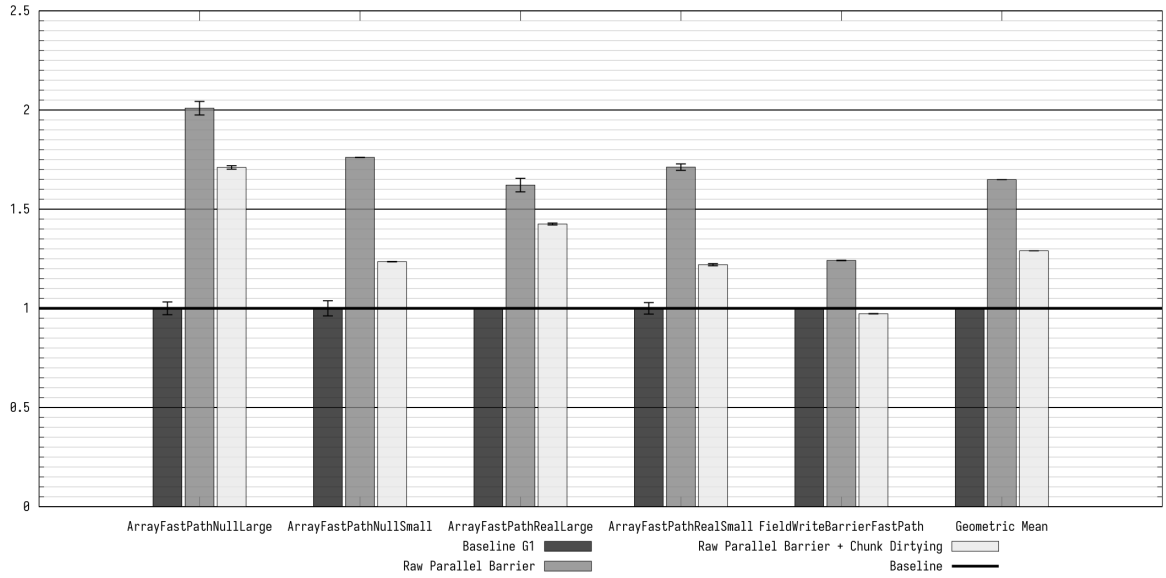


Figure 30: WriteBarrier microbenchmark results for the barrier with chunk table modification

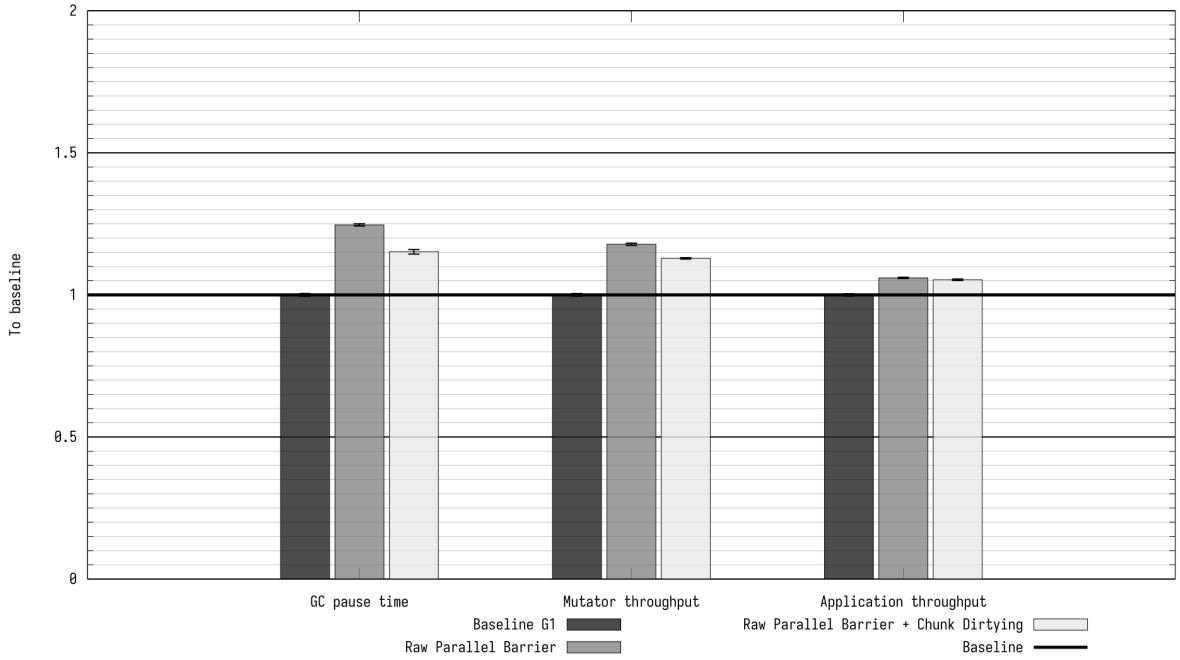


Figure 31: BigRamTester benchmark results for the barrier with chunk table modification

Benchmark	**Mean**	99% **Error**
Baseline G1		
ArrayFastPathNullLarge	1.0000	0.0322
ArrayFastPathNullSmall	1.0000	0.0382
ArrayFastPathRealLarge	1.0000	0.0016
ArrayFastPathRealSmall	1.0000	0.0290
FieldWriteBarrierFastPath	1.0000	0.0009
Geometric Mean	1.0000	
"Raw Parallel" variant		
ArrayFastPathNullLarge	2.0092	0.0343
ArrayFastPathNullSmall	1.7612	0.0006
ArrayFastPathRealLarge	1.6215	0.0342
ArrayFastPathRealSmall	1.7120	0.0161
FieldWriteBarrierFastPath	1.2415	0.0011
Geometric Mean	1.6491	
"Raw Parallel" variant + chunk dirtying		
ArrayFastPathNullLarge	1.7103	0.0088
ArrayFastPathNullSmall	1.2353	0.0011
ArrayFastPathRealLarge	1.4251	0.0049
ArrayFastPathRealSmall	1.2201	0.0054
FieldWriteBarrierFastPath	0.9728	0.0009
Geometric Mean	1.2902	

Table 17: Throughput of WriteBarrier microbenchmark with the chunk dirtying barrier

Measure	Mean	5th Percentile	95th Percentile
Baseline G1			
GC pause time	1.000	0.995	1.005
Mutator throughput	1.000	0.995	1.005
Application throughput	1.000	0.996	1.004
Raw Parallel Barrier			
GC pause time	1.246	1.242	1.250
Mutator throughput	1.178	1.174	1.182
Application throughput	1.060	1.058	1.062
Raw Parallel Barrier + Chunk Dirtying			
GC pause time	1.152	1.144	1.160
Mutator throughput	1.129	1.126	1.131
Application throughput	1.053	1.051	1.056

Table 18: BigRamTester benchmark results for the chunk dirtying barrier

Appendix F

Dynamically-switched barrier evaluation

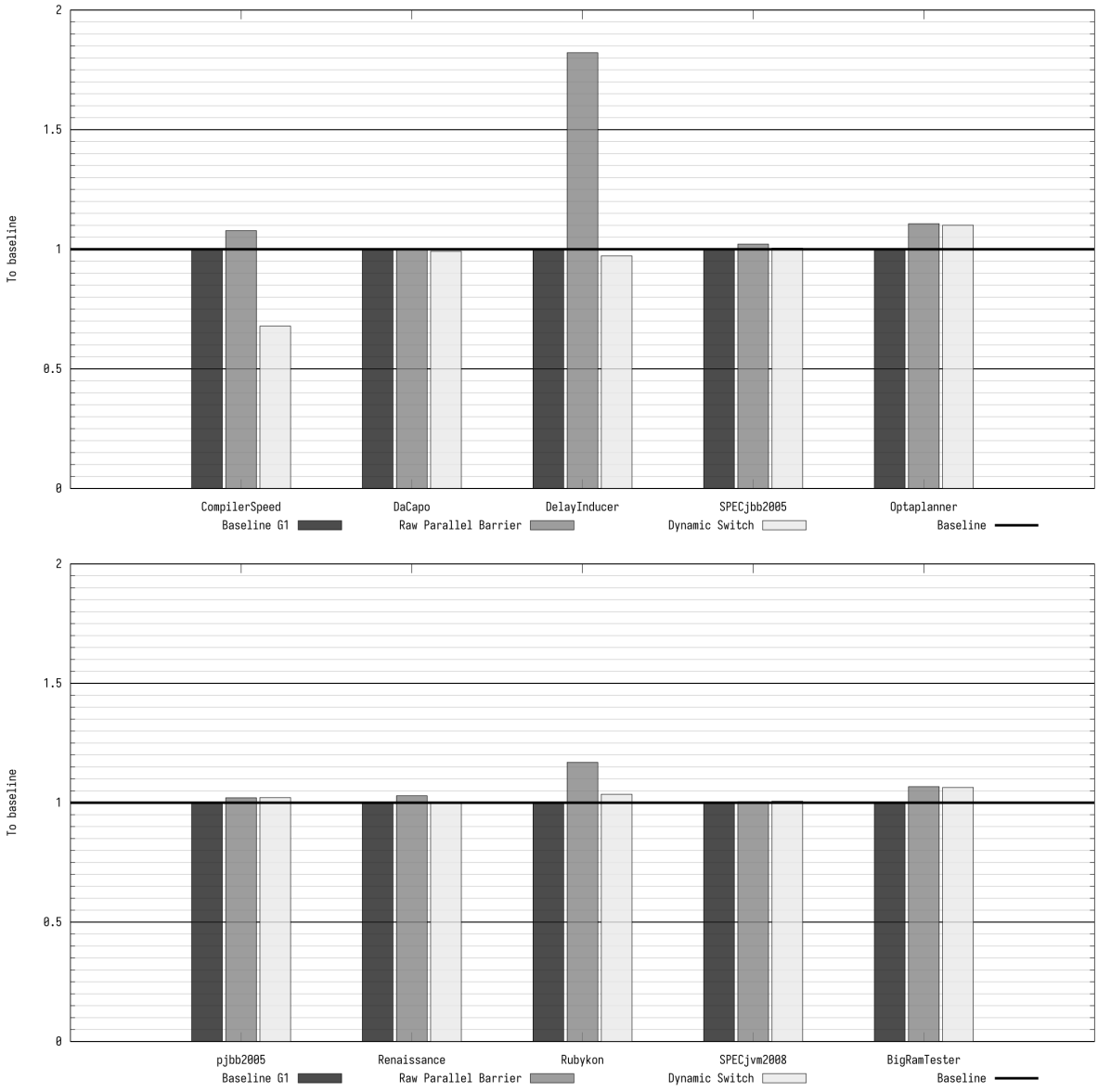


Figure 32: Throughput results of benchmark suite for dynamically-switched barrier

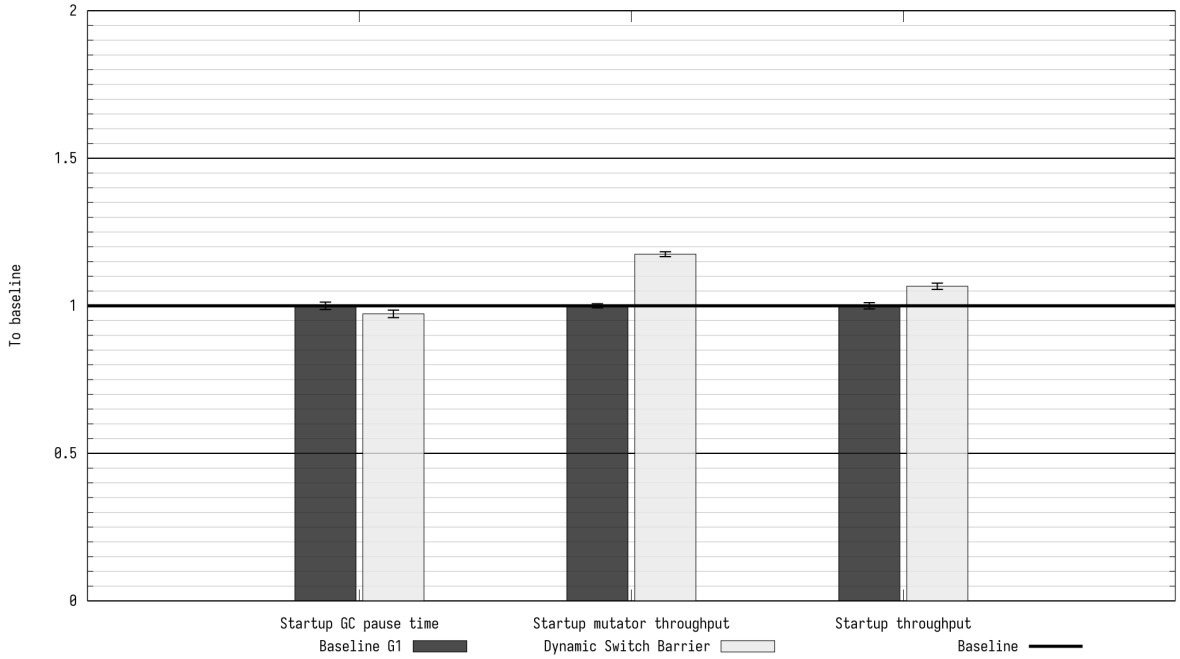


Figure 33: Startup performance results of BigRamTester for dynamically-switched barrier

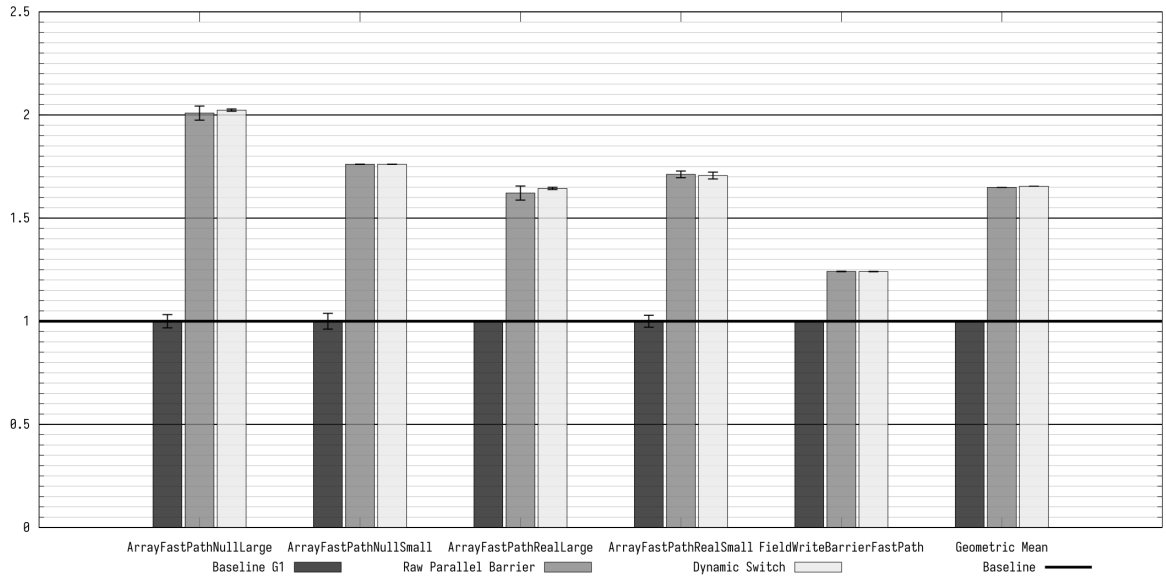


Figure 34: WriteBarrier microbenchmark results for dynamically-switched barrier

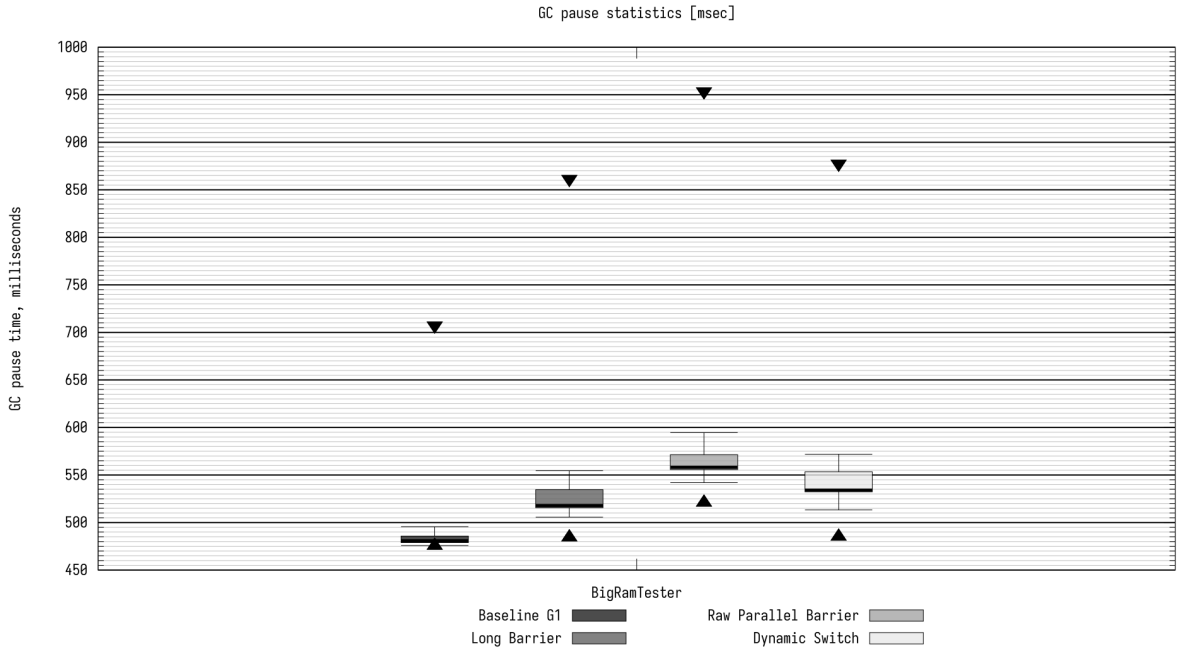


Figure 35: BigRamTester garbage collection pause time statistics for dynamically-switched barrier

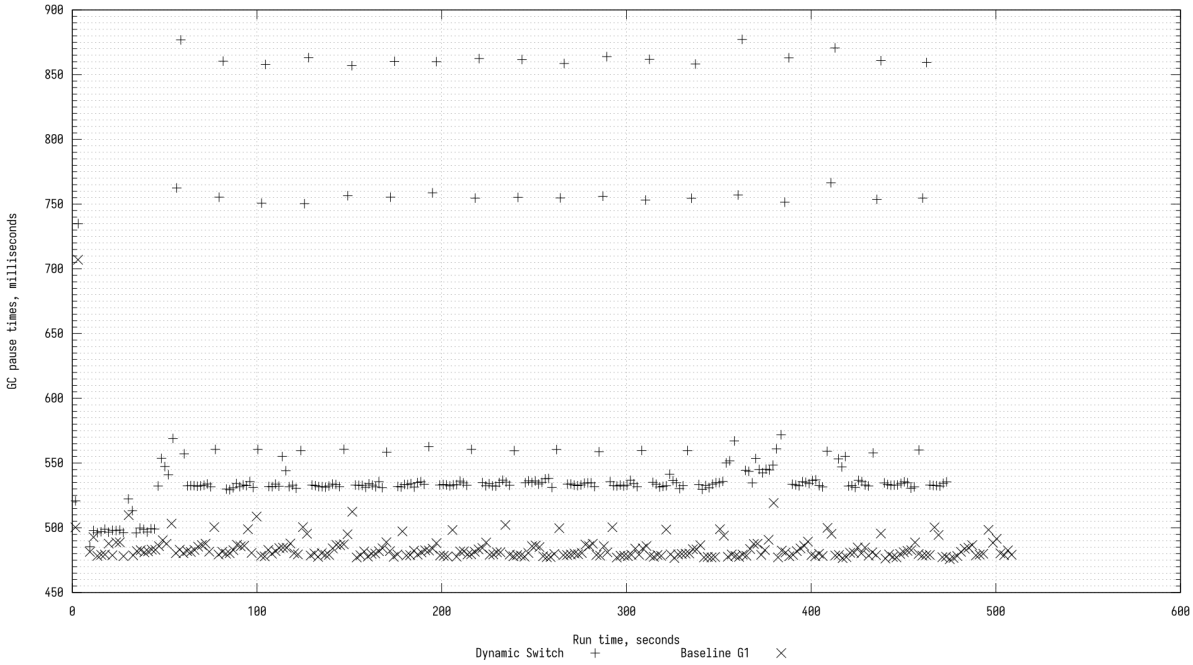


Figure 36: BigRamTester garbage collection individual pause times for dynamically-switched barrier

Benchmark	Baseline G1	"Raw Parallel" variant	Dynamically-switched barrier
CompilerSpeed	1.0000	1.0778	0.6789
DaCapo	1.0000	1.0020	0.9915
DelayInducer	1.0000	1.8216	0.9724
SPECjbb2005	1.0000	1.0215	1.0049
Optaplanner	1.0000	1.1067	1.1005
pjbb2005	1.0000	1.0207	1.0215
Renaissance	1.0000	1.0291	1.0016
Rubykon	1.0000	1.1691	1.0355
SPECjvm2008	1.0000	1.0041	1.0067
BigRamTester	1.0000	1.0672	1.0640

Table 19: Mean throughput of benchmark suite with the dynamically-switched barrier

Measure	Mean	5th Percentile	95th Percentile
Baseline G1			
GC pause time	1.0000	0.9875	1.0125
Mutator throughput	1.0000	0.9930	1.0070
Application throughput	1.0000	0.9894	1.0106
Dynamically-switched barrier			
GC pause time	0.9727	0.9600	0.9854
Mutator throughput	1.1749	1.1666	1.1831
Application throughput	1.0661	1.0555	1.0768

Table 20: Startup performance results of BigRamTester benchmark for dynamically-switched barrier

Benchmark	Mean	99% Error
Baseline G1		
ArrayFastPathNullLarge	1.0000	0.0322
ArrayFastPathNullSmall	1.0000	0.0382
ArrayFastPathRealLarge	1.0000	0.0016
ArrayFastPathRealSmall	1.0000	0.0290
FieldWriteBarrierFastPath	1.0000	0.0009
Geometric Mean	1.0000	
"Raw Parallel" variant		
ArrayFastPathNullLarge	2.0092	0.0343
ArrayFastPathNullSmall	1.7612	0.0006
ArrayFastPathRealLarge	1.6215	0.0342
ArrayFastPathRealSmall	1.7120	0.0161
FieldWriteBarrierFastPath	1.2415	0.0011
Geometric Mean	1.6491	
Dynamically-switched barrier		
ArrayFastPathNullLarge	2.0235	0.0054
ArrayFastPathNullSmall	1.7610	0.0004
ArrayFastPathRealLarge	1.6440	0.0055
ArrayFastPathRealSmall	1.7063	0.0166
FieldWriteBarrierFastPath	1.2409	0.0010
Geometric Mean	1.6547	

Table 21: Throughput of WriteBarrier microbenchmark for dynamically-switched barrier

Variant	Mean	Median	-1.5 IQR	+1.5 IQR	Min	Max	Total
	Milliseconds						
Baseline G1	484.2569	480.7530	475.7560	495.5820	475.7560	707.0220	117674.4260
"Long" variant	560.7872	517.8450	505.6840	554.5220	484.5990	861.1800	136271.2900
"Raw Parallel" variant	605.7378	558.1280	541.9700	594.6250	521.1840	953.3130	147194.2820
Dynamically-switched barrier	575.8991	534.0750	513.2510	571.8530	485.3860	877.2350	139943.4770

1.5 IQR — bounds of $\pm 1.5 * IQR$ range; corresponds to the whiskers on pause time plots.

Table 22: Pause times of BigRamTester benchmark with dynamically-switched barrier