Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services

Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, and Haibing Guan, *Shanghai Jiao Tong University;* Sanhong Li, Chuansheng Lu, and Tongbao Zhang, *Alibaba*

https://www.usenix.org/conference/atc20/presentation/wu-mingyu

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services

Mingyu Wu[†‡], Ziming Zhao[†‡], Yanfei Yang[†‡], Haoyu Li[†‡], Haibo Chen[†‡], Binyu Zang[†‡], Haibing Guan[†‡], Sanhong Li[◇], Chuansheng Lu[◇], Tongbao Zhang[◇]

[†]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
[‡]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
[◇]*Alibaba Group*

Abstract

The service-oriented architecture decomposes a monolithic service into single-purpose services for better modularity and reliability. The interactive nature, plus the fact of running inside a managed runtime, makes garbage collection a key to the reduction of tail latency of such services. However, prior concurrent garbage collectors reduce stop-the-world (STW) pauses by consuming more CPU resources, which can affect the application performance, especially under heavy workload.

Based on an in-depth analysis of representative latency-sensitive workloads, this paper proposes *Platinum*, a new concurrent garbage collector to reduce the tail latency with moderate CPU consumption. The key idea is to construct an isolated execution environment for concurrent mutators to improve application latency without interfering with the execution of GC threads. *Platinum* further leverages a new hardware feature (i.e., memory protection keys) to eliminate software overhead in previous concurrent collectors. An evaluation against state-of-the-art concurrent garbage collectors shows that *Platinum* can significantly reduce the tail latency of real-world interactive services (by as much as 79.3%) while inducing moderate CPU consumption.

1 Introduction

Today's cloud environment is an enormous beast with quantities of interconnected machines. To tame the beast, developers (1) break the traditional monolithic applications into small and interactive *services* and (2) implement services atop managed languages (such as Java, C#, and Go) to build reliable, elastic and efficient systems. Unfortunately, a tension exists between those services and managed languages. Services are designed as single-purpose and interactive applications to achieve low latency. It typically takes several milliseconds and even sub-millisecond to complete a request in interactive services. Meanwhile, managed languages like Java introduce garbage collections (GC) to manage memory resources automatically. However, mainstream garbage collectors used in interactive services will introduce stop-the-world (STW) events, where all application threads (known as *mutators*) are forced to pause so that GC threads can scan the heap for memory reclamation. The pause time is tens to hundreds of milliseconds, which are usually one to two magnitudes of the execution time for a single request in interactive services. Therefore, STW pauses will affect the latency of services and cause the long tail problem. Prior work has observed that STW pauses have significant effects on tail latency in latency-sensitive scenarios [19, 40].

There are mainly two ways to reduce the STW pause time. *Partially-concurrent collectors*, such as G1 [11] and CMS, suggests reducing STW pauses by restricting the size of *collection set* in which objects need to be collected. However, this solution will introduce more frequent collections and larger accumulated STW pause time. *Mostly-concurrent collectors*, such as Shenandoah [13] and ZGC [33], allow mutators to run in nearly all GC phases. Those collectors are quite effective in reducing the duration of STW pauses, but it requires GC threads to run constantly and spend more computing resources coordinating with mutators. In summary, both kinds of collectors achieve shorter STW pauses by occupying more CPU slices and thereby put more pressure on mutators. When the workload becomes stressful, spending more computing resources in GC may end up with even worse application latency.

In this paper, we present a new garbage collector which (1) reduces the tail latency of interactive services and (2) induces moderate CPU consumption. To achieve this, we first study the effect of GC on various latency-sensitive interactive service scenarios, including production traces in Alibaba, whose business applications rely heavily on JVM. We further analyze state-of-the-art collectors and uncover their problems respectively: *idle computing resources* in stop-the-

world pauses of partially-concurrent collectors and *considerable runtime overhead* in mostly-concurrent collectors. We then describe the *skewed memory write* behaviors of interactive service applications. We also discuss the development of hardware to show opportunities for a brand-new design.

According to the analysis, this paper proposes *Platinum*, which finds a sweet-spot in the design of concurrent collectors. It leverages idle cores in STW pauses and grants them to mutators to solve the idle computing resources problem. It then provides an isolated execution model to minimize the synchronizations between GC threads and mutators to reduce the runtime overhead in prior mostly-concurrent collectors. It further exploits recently-released hardware features (MPK) to eliminate *barriers*, the primary source of software overhead in traditional mostly-concurrent collectors. With *Platinum*, GC threads and mutators can run together with little interference with each other, so the latency and CPU utilization are both satisfying.

Platinum is implemented atop the Parallel Scavenge Garbage Collector (PSGC), a STW and throughput-oriented collector used by default in the HotSpot JVM of OpenJDK 8. Evaluation of various interactive service benchmarks show that *Platinum* significantly improves the tail latency of applications (by up to 79.3% for 99th percentile latency) compared with other concurrent collectors while preserving moderate CPU utilization.

The contributions of this paper include:

- A comprehensive analysis of latency-sensitive interactive services, including simulated industrial workload with production traces, to understand the effect of GC (Section 2).
- *Platinum*, a concurrent garbage collector that can reduce the tail latency for interactive services while preserving moderate CPU consumption (Section 4).
- Experiments on different latency-sensitive scenarios to confirm that *Platinum* can outperform other garbage collectors on tail latency and CPU utilization for interactive services (Section 5).

2 Analysis: when interactive services meet GC

In this section, we will analyze the effect of GC in interactive services with production traces in Alibaba.

2.1 A page is multiple services

Alibaba has one of the world's largest e-commerce platforms. To serve an ocean of concurrent requests from a vast number of clients at any time, Alibaba has deployed its platform atop a large scale of machines. Traditional monolithic applications are too cumbersome and thus not suitable to be distributed to many machines due to the prohibitive cost of development and maintenance, so the developers from Alibaba have split them into smaller, simple-purpose, and interactive units, namely *services*. A service is much smaller to simplify the deployment process, and it can also be replicated to enhance the availability of the overall platform.

Due to the complex business logic in Alibaba, every operation from users require the collaboration of various services. For example (shown in Figure 1), when a user wants to check out, she will request for the check-out page, where all items in the shopping cart (cart service) will be combined together so that the most cost-effective way to purchase them with available coupons will be automatically computed (coupon service). In addition, the check-out page also recommends other items according to those in the cart and the user's prior purchase behaviors (recommendation service). Those services also interact with each other, and they may communicate with the cache service for high-speed data fetching.

Since those services are mostly written in Java for reliability, productivity, and compatibility, all of them will be affected by GC. Alibaba has provided some workarounds to mitigate the effect of GC. For example, when the recommendation service is not responsive, the page render is capable of generating a simplified web page without any recommendation information for users. Unfortunately, not all services can benefit from those optimizations. For example, as for the coupon service, the latest-available coupons must be included for computation. Otherwise, the users will fail to purchase in the most cost-effective way. We have conducted a series of tests to understand the role GC plays in those latency-critical services. Note that a garbage collector usually includes *minor GC* and *major GC*. The minor GC collects a part of the heap while the major GC collects the whole heap. Since major GC rarely happens in the scenario of interactive services, this work will focus on the effect of minor GC.

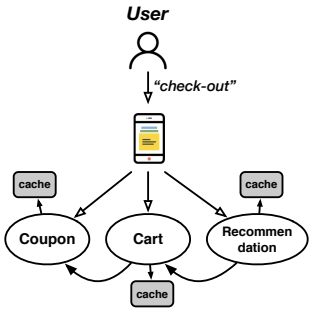


Figure 1: A simplified model to shape the multi-services scenario in Alibaba

2.2 STW pauses: the culprit for tail latency

Since garbage collectors will pause application threads for memory reclamation, it will significantly affect the response time of requests in interactive services. To better understand the effect of GC, we leverage a simulated online environment for analysis. The simulated environment is built with 120 service instances, each of which is deployed in a container. All services are unmodified applications extracted from the real industrial workload in Alibaba. During testing, those services will be fed with requests at a given throughput. In our

setting, the cluster will handle 200 user requests per second, and this value is chosen to stress the coupon services. All requests are traces collected from the production environment. The default garbage collector for services is CMS (Concurrent Mark Sweep [28]), a classic concurrent collector introducing relatively long STW pauses.

To understand the relationship between application behavior and GC pause time, we add a new Java option -*XX:InstrumentedPauseTime* to the vanilla OpenJDK 8. When the value is not zero, JVM will add a *sleep* call at the end of GC to extend the GC pause time. The duration of sleeping time can be adjusted by modifying the value of -*XX:InstrumentedPauseTime*. We are mainly interested in the coupon service as it is latency-sensitive. Our findings are listed below.

Stop-The-World (STW) pauses is a killer factor for the tail latency. We exploit a scatter plot in Figure 2 to illustrate the relationship between GC and request latency. Points in the scatter plot stand for the completion time of a request (measured in the server-side), while red vertical lines stand for the start time of GC. As Figure 2 suggests, each GC cycle will follow some stragglers, which significantly affect the tail latency. Although the request latency could be influenced by many factors such as network, disk I/O, and other collaborative services, GC is the one to dominate the tail latency. Note that this experiment actually underestimates the effect of GC on the request latency, as the statistics are collected from servers, which overlook the queuing time in the client-side. One can expect more requests are affected by GC if each request can be tracked in the upstream services, which is unfortunately not supported in the cluster environment.

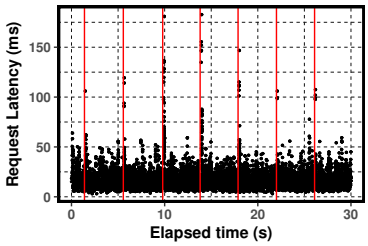


Figure 2: The relationship between STW pauses and request latency in the coupon service. The collector is CMS

GC pause time shows a super-linear relationship with tail latency. During the evaluation, we change the value of -*XX:InstrumentedPauseTime* for all five coupon service instances from 0 to 10ms and collect the log from them. Figure 3 shows the change in the request latency for a coupon service instance in a cumulative distribution function (CDF) form, compared with that in the vanilla setting. The result suggests that the increased pause time has a magnified impact on the tail latency: When the GC pause is added by 10 ms, the 99th percentile latency is increased by 11.435 ms, and the 99.9th percentile latency is enlarged by 32.950 ms. It is because requests in clients are generated at any time, regardless of the running state of services. When the coupon service instance is undergoing a garbage collection, the pending requests will gradually increase and queue up. Once GC ends, a pending request cannot be processed until the preceding ones are finished. If GC pauses are extended, those newcomers must wait for not only a longer pause but also more pending requests, so the tail latency will increase faster than GC pauses.

In the real-world cooperative multi-services scenario (such as the check-out case in Alibaba), the problem can be deteriorated because all participating services may be affected by GC. For example, Mass et al. [25] have observed that Cassandra replicas on GC will hinder the whole storage system from constructing a quorum, which leads to prohibitive user-experienced latency. Therefore, GC pauses is a key factor for tail latency reduction in interactive services.

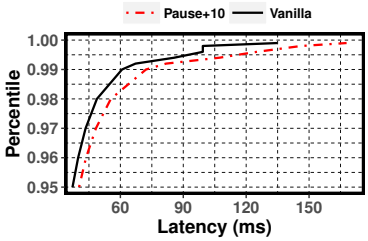


Figure 3: The CDF of request latency for the coupon service atop CMS

2.3 Is concurrent GC helpful?

Due to the detriment of STW pauses, interactive services usually adopt *concurrent GC* for better application latency. Concurrent GC can be roughly divided into two categories: *partially-concurrent collectors* and *mostly-concurrent collectors*. Partially-concurrent collectors, such as CMS (mentioned above) and G1 [11], allow the co-execution for mutators and GC threads in only some phases of GC. Therefore, they still introduce STW pauses, and they provide solutions to further reduce them. *Mostly-concurrent collectors*, such as ZGC [33] and Shenandoah [13], nearly eliminate STW pauses so that mutators can run constantly. We study those two kinds of collectors on interactive services respectively.

Partially-concurrent GC. As shown in Figure 2, the tail latency problem exists in partially concurrent collectors like CMS since they still pause mutators for collection. Fortunately, partially-concurrent collectors usually provide options to adjust the duration of pauses. For example, CMS allows users to adjust the size of the heap area required to be collected, while G1 can be restricted with a pre-assigned maximum pause time. After setting those options, partially-concurrent collectors will adjust the heap layout to meet the requirement.

We exploit G1 to study the effect of those adjustable options. G1 is a highly-tunable partially-concurrent collector which becomes the default one since OpenJDK 9. It is a generational collector whose heap space consists of *young space*

and *old space*. Its GC algorithm contains three parts: *minor GC* on the young space, *mixed GC* on the young space and a part of the old space, and *major GC* on the whole heap. The minor GC, which is stop-the-world, happens the most frequently. To reduce the duration of pauses, G1 provides an option *-XX:MaxGCPauseMillis* for users to control them. Therefore, we launch the coupon service on G1 by setting the option to various values (30ms, 40ms, 60ms) and evaluate the performance respectively. This evaluation is single-point, where only one coupon service is launched to process requests. With the single-point evaluation, the latency can be accurately collected from the client-side, including the aforementioned queuing delay. The requests are sent in a fixed throughput to simulate a stressful scenario (4000 requests per second in our setting), and they are still real-world traces extracted from the online environment in Alibaba. The duration for data collection is a minute.

Table 1 shows the statistics on GC and application for different settings. With smaller *MaxGCPauseMillis*, both the minimum and the average GC pause time are reduced. However, young GC is triggered much more frequently. It is because G1 controls the GC time by tuning the size of the young space, which serves for memory allocation requests from mutators. Since young GC is triggered when the memory resource in the young space is exhausted, its frequency will increase when the young space shrinks. As a result, although the per-GC pause time is cut down by lowering *MaxGCPauseMillis*, the overall time consumed by GC grows larger. With larger overall GC pause time, more application requests are affected, and less computing resource is available for mutators. Therefore, the tail latency problem is not resolved but becoming much more severe: the p99 latency with the 30ms setting is 13X of that with 60ms. Besides, the average CPU utilization in the 30ms setting is also increased by 15.3% compared with the 60ms setting. This experiment suggests that partially-concurrent GC achieves shorter pauses by consuming more CPU resources, which may end up with worse tail latency.

Metrics	30ms	40ms	60ms
Minimum GC pause (ms)	21.815	21.459	39.856
Average GC pause (ms)	34.441	40.724	48.491
The number of GC	550	392	111
p99 latency (ms)	1942.09	1389.99	148.85
Average CPU utilization	51.45%	50.81%	36.17%

Table 1: The statistics on GC and the coupon application with different settings of G1GC

Mostly-concurrent GC. Compared with partially-concurrent GC, mostly-concurrent GC allows mutators to execute nearly all the time. Recently released mostly-concurrent collectors, like ZGC and Shenandoah, claim to have reduced GC pauses to several milliseconds regardless of the heap size. In this work, we mainly study Shenandoah[1], a mostly-concurrent garbage collector released in OpenJDK 12.

We launch the coupon service atop Shenandoah and evaluate it with the same setting as G1. After collecting the GC log, we conclude that Shenandoah is very effective in reducing the pauses, and the average pause time is only 18.764 ms. However, the latency of requests is prohibitive: the p99 latency is over 3 seconds, which is 1.86X even compared with the worst case in G1 (30 ms). We observed that the time when GC threads are active is 53.05 seconds, which means that GC threads are active nearly all the time. Meanwhile, the average CPU utilization reaches 83.05% (the peak utilization reaches 96.74%), which suggests that GC threads consume much more CPU resources than other collectors, and mutators do not have enough CPU slices to sustain such a high throughput. As a result, the p99 latency with Shenandoah is not decreased but increased when encountering stressful workload.

To conclude, both partially-concurrent GC and mostly-concurrent GC are making a tradeoff between the duration of GC pauses and the CPU efficiency. Shorter GC pauses mean that GC threads will consume more computing resources and thus affect the performance of mutators. We instead want to build a garbage collector with both short pauses and moderate CPU efficiency to support the interactive services.

3 Implications for a new GC design

Before proposing our design, we take a closer look at the designs of prior concurrent garbage collectors to uncover the opportunities to build a new garbage collector for our goals: short GC pauses and moderate CPU utilization. We then reveal the *skewed memory write* behavior in interactive service applications, which is crucial to our design. We also introduce *memory protection keys* (MPK), a recently released hardware feature, and suggest how collectors can benefit from it.

3.1 Problems in concurrent garbage collectors

According to the behavior of mutators during GC, concurrent garbage collectors can be roughly divided into two categories. However, both of them have problems hindering them from achieving both satisfying GC pause time and CPU efficiency.

Idle computing resources for partially-concurrent collectors. STW pauses in partially-concurrent collectors require all mutators to suspend and leverage all computing resources to collect objects. This design avoids interferences between mutators and GC threads, but it also results in idle cores during GC due to its scalability issues.

There are two reasons why garbage collectors cannot scale well. First, the collection algorithm is somewhat similar to

[1]We exploit Shenandoah as a baseline in this paper as it provides backward support to OpenJDK 8, a long-time-support version which is widely leveraged in both industries (like Alibaba) and open-sourced projects.

graph traversal: GC threads will start from some *root objects* and mark all reachable ones through references among objects. The traversal is highly unpredictable as we do not know how many references a thread will process in advance. Therefore, collectors are prone to load-imbalance and often turns to work-stealing to achieve dynamic balancing. However, work-stealing is also ineffective in that it has to search for tasks from all other threads and contend with others during task fetching. Prior work [37] shows that work-stealing even causes performance slowdown in extreme cases. Second, the scalability of collectors is affected by many factors. Recent studies have shown that the NUMA architecture [17], synchronization protocols [16], and even the Linux scheduling mechanism [38, 40] could have a significant influence on the GC scalability. Therefore, it is difficult to come up with a general algorithm which is scalable for various scenarios.

Given those reasons, partially-concurrent collectors exploit a conservative mechanism where the number of GC threads is smaller than that of cores. In OpenJDK, the number of GC threads by default is about five-eighths of the total core count. This policy also works for STW collectors like PSGC. The developers of OpenJDK explain their choice in the comment of the source code: *For very large machines, there are diminishing returns for large numbers of worker threads. Instead of hogging the whole system, use a fraction of the workers for every processor after the first 8.* This default setting mitigates performance degradation as the number of cores increases and has been used in prior GC studies [40]. However, it also results in idle cores during collection. Instead of searching for a perfectly scalable collection algorithm, we want to introduce some concurrent mutators to leverage those idle cores while still preserving the performance of collectors. Note that introducing concurrent mutators during GC will not hurt the overall CPU efficiency much, as the duration of STW pauses only occupies a very small portion of the whole application's execution time in interactive services.

Considerable runtime overhead for mostly-concurrent collectors. Unlike partially-concurrent collectors, mostly-concurrent ones allow mutators to run simultaneously with GC threads nearly all the time to reduce the application latency. However, GC threads and mutators must synchronize with each other as they may modify the same objects simultaneously, which introduces overhead for both GC threads and mutators. Besides, mostly concurrent collectors further introduce *barriers* in mutators for GC invariant checking. A barrier is a piece of code instrumented before specific instructions. For example, Shenandoah exploits *read barriers* for mutators, meaning that mutators need to check the invariants for every single read operation on references, no matter if GC is active. Those two factors introduce significant runtime overhead to the runtime. Therefore, although Shenandoah has been optimized through aggressive Just-In-Time (JIT) compilation, it still causes a 24% slowdown for real-world workload compared with G1 [13].

3.2 The skewed memory write behavior

Memory behavior of applications is quite crucial to GC performance and thereby affect the design of collectors. For example, the memory behavior of big-data processing workload is at odds with assumptions in traditional GC algorithms and stimulates a series of big-data-friendly garbage collectors [18, 31]. To this end, we study the memory behavior of latency-sensitive services to explore new space for garbage collector design.

Session-based execution model. Since services are interactive, their execution can be divided into many *sessions*. The service will process a request when a session starts and generate a response before the session ends. Sessions are mostly isolated from each other: a session will not try to access the objects created by other sessions unless those objects are globally visible through shared data structures. Therefore, we presume that the memory behavior of those session-based applications will be skewed, i.e., *the memory accesses within a session will fall into a very small range where the session allocates memory.* The memory range is referred to as *working set* in this paper. We have conducted an in-depth analysis to validate this hypothesis.

Memory behavior analysis. To analyze the memory behavior of those session-based applications, we first need to demarcate all sessions. We add two new JVM calls, *Sessionbegin* and *Sessionend*, to achieve this goal. After *Sessionbegin* is invoked, the JVM will track all memory the session allocates. When *Sessionend* is called, the JVM will print out the size of the overall allocated memory, which is the working set size for the current session.

Our hypothesis is that session-based interactive services share similar memory behavior. To confirm this, we have also studied three other applications: *SpecJBB2015*, a simulated online supermarket, *Cassandra* [9], a key-value store, and *ShopCenter*, another interactive service used in Alibaba. All of them will process requests from clients in sessions and send responses back. Note that both ShopCenter and Coupon leverage production traces for analysis. To profile their memory behavior, we instrument *Sessionbegin* right before the request is processed and *Sessionend* when the processing finishes.

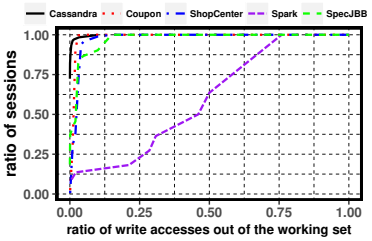


Figure 4: The CDF for the memory range of write accesses in various applications

Figure 4 shows the CDF curve of memory writes for three different interactive applications. As for the coupon service, over 99.5% sessions have less than 3% writes out of their working sets. The working set of a session usually spans several megabytes, which is quite small compared to the Java heap (typically tens to hundreds of gigabytes). The other applications share similar memory behaviors. For Cassandra, even the worst case has 72.4% writes inside its own working set. Other kinds of applications, however, do not follow this behavior. As for Spark, a big-data processing framework, 50% sessions (a session in Spark is defined as the process the worker handles a task assigned by the master) have more than 46% writes out of their working sets.

This experiment confirms our hypothesis that memory writes in sessions of interactive services are skewed. The skewed memory write behavior opens up an opportunity for optimization: since mutators mostly update objects inside their working sets, they can run simultaneously aside GC threads with rare interference if GC threads avoid reclaiming their working sets. Since previous garbage collectors do not put the skewed memory write behavior into consideration, this finding motivates us to design a new garbage collector to guarantee both satisfying latency and CPU utilization.

3.3 MPK and Garbage Collectors

Since barriers in concurrent GC are costly due to its high execution frequency, some collectors [2, 4, 10] have turned to a virtual-memory-based mechanism for a transparent and efficient solution. The virtual-memory-based mechanism leverages the access permissions on the page table entries and relies on hardware to check the invariants. For example, a GC thread can mark a virtual page as *being-collected* by setting its permission as read-only. Mutators writing to this page will trigger a page fault and execute synchronization-related operations in a pre-registered handler. GC threads working on other pages will have no overhead as no page fault is triggered. This method can eliminate the need for the barrier code, and it can also be used in other areas, such as software transactional memory [1].

Unfortunately, threads in a process will share the same permissions on all page table entries. In the previous example, if a GC thread also wants to modify the being-collected page, which should be legal, it still suffers from a page fault. Those *false page faults* impede collectors to implement an efficient protection mechanism.

Intel MPK (Memory Protection Keys) is a hardware feature available recently in the SkyLake server CPUs. With MPK, users can categorize virtual pages into different *domains*, which will be denoted with special bits in the corresponding page table entries. Furthermore, they can manually grant different permissions for each domain to different threads through a special register. This hardware feature makes it possible to support finer-grained *thread-level isolation* by adjusting the permissions over domains for each thread. Previous work has studied the usage of MPK in the security area [14, 20, 35, 43], but it could also be leveraged for performance consideration.

4 Design

According to our analysis, we build *Platinum* for both satisfying latency and CPU efficiency.

4.1 Overview

Platinum is built atop PSGC, a STW garbage collector in OpenJDK. Compared with concurrent collectors, PSGC is more CPU-efficient as it always pauses mutators during GC to achieve maximum collection throughput. Nevertheless, it still results in idle cores due to scalability issues. *Platinum* inherits the generational design from PSGC to divide the heap into *young space* and *old space*. The young space further consists of three sub-spaces. The *eden-space* (usually the largest sub-space) serves for allocation, while the other twos, *from-space* and *to-space*, are used to store objects surviving at least one GC cycle. Only objects having lived for long will be copied into *old space*. It contains two GC algorithms: minor GC for the young space and major GC for the whole heap. *Platinum* mainly considers the minor GC while leaving the major GC as future work.

Figure 5 illustrates the infrastructure of *Platinum*. The design highlights of *Platinum* include:

Sufficiently leveraging computing resources. *Platinum* collects the cores not used by GC threads and grants them to mutators for better application latency, hence resolving the idle computing resources problem in partially-concurrent GC.

Isolated execution between GC threads and mutators through heap partition. According to the skewed memory behavior in interactive services, *Platinum* partitions the heap space so that GC threads and mutators can focus on processing objects in different parts of the heap. This design minimizes the synchronizations between GC threads and mutators, so the coordination overhead in mostly-concurrent collectors is greatly reduced in *Platinum*.

Hardware-assisted barrier elimination. *Platinum* leverages the MPK hardware feature to remove the need for software barriers adopted in prior mostly-concurrent collectors. This design further reduces the runtime overhead and improves CPU efficiency. *Platinum* also exploits the RTM feature to ensure the atomicity of write operations in mutators.

4.2 Platinum in steps

Similar to the minor GC algorithm in PSGC, *Platinum* is copy-based: GC threads will simultaneously scan the young space for live objects and copy them to their new address. The process of *Platinum* GC is mainly in three steps:

1. Initial marking. This step is somewhat similar to the initial marking pause in G1 [11]. In the initial marking phase of *Platinum*, GC threads will scan the runtime stack of muta-

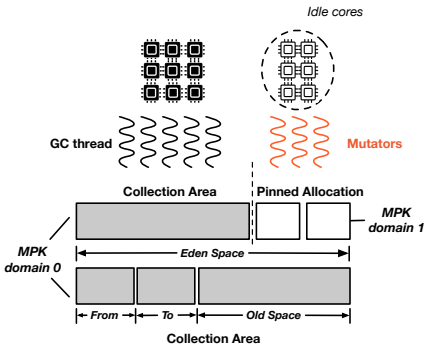


Figure 5: Overview of *Platinum*

tors to mark live objects. Those live objects on the stack will be treated as *root objects* for further object copying. This step is stop-the-world because the values on the stack are changed quite frequently, and we want a stable state at the beginning of a collection. It usually takes only a few milliseconds to finish initial marking.

2. Concurrent scavenge. After initial marking, *Platinum* will invoke mutators to resume their execution while GC threads will concurrently scan the heap to identify and copy live objects. Thanks to the isolated execution mechanism (detailed in Section 4.4), GC threads and mutators will focus on processing objects in different areas of the heap and hardly interfere with each other. Therefore, GC threads can directly copy live objects without considering the behavior of mutators. This step occupies the most time of the whole GC process in *Platinum*.

3. Stop-the-world scavenge. When GC threads have finished their work, *Platinum* will pause running mutators again. Since some objects in mutators' working sets are not processed by GC threads in the concurrent scavenge step, they may contain "stale" references to objects which have been evacuated. Therefore, *Platinum* should scan those objects for correctness guarantee. Since the number of objects is quite small, this step will not take long.

With the above three steps, *Platinum* can: (1). improve the application latency because mutators are allowed to run in most time of GC; (2). avoid consuming too much CPU resources because GC threads are isolated from mutators to retain satisfying collection throughput. Therefore, *Platinum* can take into account both latency and CPU efficiency at the same time. We will introduce the core designs of *Platinum* in the rest of this section.

4.3 Idle core collection

To make *Platinum* effective, the application should configure the number of GC threads to be smaller than that of cores (or directly adopt the default setting in OpenJDK). When *Platinum* is initialized, it will automatically bind the GC thread into different CPU cores. Other cores with no GC threads running will be remembered by *Platinum* as *idle cores*.

When *Platinum* is not active, mutators are free to run on any cores. When GC is triggered, *Platinum* will constrain mutators to only choose idle cores for execution to avoid contending computing resources with GC threads. This is achieved by setting the affinity values of mutators with the *sched_setaffinity* interface in Linux. Those affinity values will be reset at the end of GC. This design ensures not only idle cores are sufficiently used by mutators but also each GC thread monopolizes its assigned core.

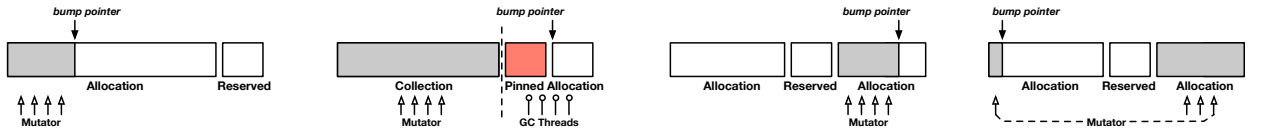
4.4 Isolated execution with heap partition

To achieve isolated execution between GC threads and mutators, *Platinum* partitions the heap into three *areas* during GC (shown in Figure 5). The first area, namely *collection area*, will be collected by GC threads. The collection area covers the most part of Java heap, including from-space, to-space, old-space, and the largest part of eden-space.

The other two areas, in contrast, are used by mutators and thus not collected in this GC cycle. Since write operations of mutators fall into a very small range (see Section 3.2), we use the *pinned area* to include objects which are highly possible to be modified by mutators in the near future. The last part is the *allocation area*, which is used by mutators to allocate new objects during garbage collection. Those newly allocated objects should be considered alive and only be scanned in the next collection cycle.

Figure 6 illustrates how the three areas work in *Platinum*. During normal execution, *Platinum* will partition the eden-space into two areas (Figure 6a). The larger one will serve memory allocation requests from mutators while the smaller one will be reserved. *Platinum* also adopts a *bump pointer* to denote how much memory has been used.

When the bump pointer reaches the end of the allocation area, GC will be triggered, and GC threads will become active. In the initial marking phase, *Platinum* will partition the eden-space into the three areas mentioned before (Figure 6b). The reserved area in the normal execution will become the *allocation area* where concurrent mutators create new objects during GC. The larger one will be further split into *collection area* and *pinned area*. GC threads will only collect the collection area while mutators mainly modify the pinned area and the allocation area. Compared with the collection area, the pinned area resides close to the bump pointer. This design choice is based on the memory allocation mechanism in JVM: each mutators will first allocate a *segment* from the global heap, and then allocate memory from the segment until it is filled up. Therefore, *Platinum* locates the pinned area adjacent to the allocation area to contain the latest segments allocated by different threads. The size of the pinned area is preset to a fixed proportion of the whole eden-space. The default value is 1/128, which can include segments from tens of mutators while inducing moderate pauses during the stop-the-world scavenge step. *Platinum* also allows users to tune this value for better performance. Users can leverage the aforementioned *Sessionbegin* and *Sessionend* API to cal-



(a) The heap space is divided into two areas during normal execution

(b) GC starts: partition the heap into three areas for isolated execution

(c) GC ends: re-partition the heap into two areas

(d) When the bump pointer reaches the end, it will "jump" to the start address for memory allocation

Figure 6: The heap layout in *Platinum*. The colored part stands for allocated memory.

culate the working set size for their applications during prerun, and modify the pinned area to a proper size accordingly.

During the concurrent scavenge step, GC threads will apply range checks to determine if an object falls into the collection area before accessing it, and only those in the collection area will be processed. Those range checks avoid the case where GC threads try to copy an object which mutators are modifying. Since the collection area is consecutive, range checks can be implemented with cheap comparison instructions and thus introduce little overhead. As a result, GC threads and mutators are enforced to concentrate on processing objects in different areas, which eliminates the need for synchronization and thereby mitigates the runtime overhead. For objects not processed during concurrent scavenge, they will be scanned and updated in the subsequent stop-the-world scavenge step.

Before GC ends, GC threads become inactive, and *Platinum* should reorganize the eden-space (Figure 6c). Since a part of the space has been occupied by live objects (pinned and allocation area), *Platinum* will mark it allocated. The rest of the space will still be partitioned into two areas, while the reserved one will reside adjacent to the pinned area in the last GC. The bump pointer will grow from the original allocation area and goes to the start address once it reaches the end (Figure 6d). When the allocation area is exhausted, a new GC cycle is activated.

4.5 Hardware-assisted barrier elimination

Heap partition restricts GC threads to only process objects in the collection area and thus reduces the coordination overhead. However, since mutators are running Java code, they are free to access any objects in the Java heap, including those in the collection area, which violates the *isolated execution* semantic. A traditional solution proposed by prior concurrent collectors is to leverage *barriers* to detect and correctly handle those operations. A barrier is a piece of code instrumented before specific operations for the interest of GC. For our problem, a garbage collector can adopt *write barriers*, which instrument range check operations before every write. As shown in Figure 7, for a field update operation ($y.x = z$), the collector should insert a range check to ensure that the address of the field ($y.x$) is not in the collection area. If the range check fails, mutators should turn to a prepared slow path (*atomic_update*) to update the field atomically to ensure correctness.

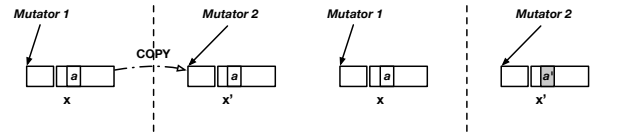
```
1  // barrier code start
2  if (in_collection_area(y.x)) {
3      atomic_update(y.x, z);
4  }
5  // barrier code end
6  else {
7      // Field update
8      y.x = z;
9  }
```

Figure 7: An example of write barriers

As mentioned in Section 3, barriers can cause significant overhead as they are instrumented with *every* specific instruction, no matter if GC is active. For write barriers, they must be executed before every write operation, including interpreter code, JIT code, and even part of native code inside JVM. We instead provide a hardware-assisted solution atop MPK to eliminate the write barriers.

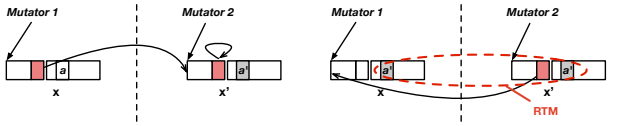
To leverage MPK, *Platinum* divides the Java heap space into two *domains*: *GC domain* and *mutator domain*. The *GC domain* only contains the collection area, while the *mutator domain* consists of the pinned area and the allocation area. When threads are initialized, they will be granted with corresponding permissions: mutators have read-write permissions to the mutator domain and read-only permissions to the GC domain; GC threads have read-write permissions for both two domains. The permissions are fixed throughout the lifecycle of a thread. On the other hand, the address ranges for those two domains are changed with the three areas dynamically. When the collector is inactive, the whole Java heap space belongs to the mutator domain so that mutators are free to access any objects. When GC starts, the collection area should be put into the GC domain. Afterward, if concurrent mutators' write operations fall into the GC domain, they will trigger page faults, and thus no software barriers are required. Thanks to MPK, *Platinum* can *automatically* detect write operations violating the isolated execution mechanism, and the control flow will be transferred to a customized handler to correctly process those operations. Although processing a page fault is more costly than executing a software barrier, the possibility of triggering a page fault in interactive services is much smaller than that for software barriers, and

the amortized overhead can be reduced.



(a) Object x has been copied to x' by GC threads, while two mutators hold a reference to different copies

(b) Mutator 1 cannot timely read the modification from mutator 2, which violates the serializability



(c) *Shenandoah* exploits an indirection pointer so that accesses to the old object will be forwarded to the new one

(d) *Platinum* instead updates both two copies atomically by putting the updates into the same hardware transaction

Figure 8: The dual-copy problem and solutions

4.6 Handling violated writes

When a write operation from a mutator falls into the collection area, the customized handler will take over and temporarily request for read-write permissions for the GC domain. This request is safe as the code in the handler is totally controlled by *Platinum*. Afterward, the handler is responsible to simulate the original write operation on the mutator's behalf by modifying the corresponding object in the collection area.

However, the simulation process must be deliberately designed as GC threads are concurrently copying those objects. If an object has been copied, the old one and the copied one will co-exist in the Java heap until GC ends. Since mutators may still have references to the old object, the consistency between the two copies must be maintained. Consider the case in Figure 8a where object x has been copied (say x'). Suppose mutator 2 gains a reference to x' and modifies a field a in x' (Figure 8b), the modification should be visible to all mutators. However, if mutator 1 still holds a reference to x and reads its content, it can only get a stale value of a. We refer to it as the *dual-copy problem*, which breaks down the serializability of the whole program.

Prior concurrent collectors have similar problems as they also allow mutators to run in scenarios where two copies of the same object are both visible in the heap. They leverage *read barriers* to force mutators always to access the newest one. The implementation of read barriers has many variants, and one of them is to use Brook's style indirection pointers [5] as Shenandoah [13] does. This solution requires adding an extra field in the header of every Java object, which stores a pointer to the newest copy of this object. As illustrated in Figure 8c, x points to x' while x' points to itself. In this way, when mutator 1 tries to access x, the indirection pointer of x will guide it to access x' instead so that it can get the updated value of a. This solution is simple and straightforward, but it introduces considerable overhead, as analyzed in Section 3.1.

Rather than using read barriers, *Platinum* guarantees correctness by updating *both* copies in the customized page fault handler. As shown in Figure 8d, *Platinum* keeps the added field in Shenandoah to store a *back pointer* to the old copy. For the old object, since the original PSGC will store a *forwarding* pointer in its header referring to the new object, the added field is useless. In the above example, when updating x', *Platinum* will locate x with the back pointer and update the value of a for both x and x'. Even though mutator 1 retains a reference to x, it can still fetch the updated content by directly reading x. This mechanism certainly doubles the write operations, but it relies on the observation that only a few write operations from mutators will happen in the collection area, so the overhead will be trivial.

Prior work like Sapphire [21] also exploits similar mechanisms, but it struggles to make the updates atomic, i.e., the updates to both copies should be visible to mutators simultaneously. Fortunately, recent hardware development has provided us with new opportunities for design. *Platinum* embraces the *Restricted Transactional Memory* (RTM) feature by Intel, which guarantees the atomicity of a piece of code by wrapping it into a hardware transaction. Since RTM requires that the working set of the transaction should be small (otherwise the abort rate will dramatically increase), we only put the update operations into the hardware transaction to construct a very small working set (less than 100 bytes). We have also prepared a fall-back handler in case the transaction fails. The handler retries the transaction in most cases, but it will try to grab a global lock if the transaction fails for many times (which happens very rarely). With RTM's help, *Platinum* can update both objects atomically and return to normal execution, which provides a strong consistency guarantee and introduces moderate overhead.

5 Evaluation

Platinum is implemented in the HotSpot JVM of OpenJDK 8u141 with about 7,500 LoCs. We leverage three various applications for evaluation:

SpecJBB2015. SpecJBB2015 is a business benchmark that simulates an online supermarket to process incoming purchasing requests. Alibaba usually exploits it as a simplified example to simulate the online production environment.

Cassandra. Cassandra is an open-sourced key-value store which is usually leveraged as a latency-sensitive application by prior work [7, 40]. We leverage YCSB as the testbed, but it is executed in a closed-loop model where a client will not send a second request until its receives the response for its last one. This model cannot reflect the fact that requests have to wait until prior ones are finished. Therefore, we have modified the execution model of YCSB to open-loop, where clients send requests in a fixed throughput regardless of the responsiveness of servers. The version of Cassandra for our

evaluation is 3.11.4.

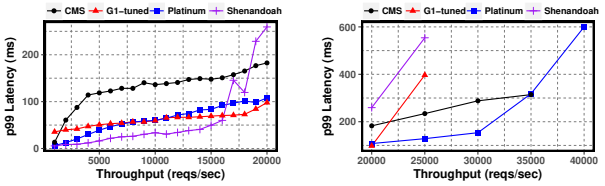
Coupon. Coupon is an online interactive service used in Alibaba, and we use it to confirm that *Platinum* actually works in real applications.

We also compare the performance of *Platinum* against other mainstream garbage collectors.

CMS. Concurrent-Mark-Sweep (CMS) is a classic partially-concurrent garbage collector. Its major GC is concurrent, and the minor GC is stop-the-world. We leave CMS untuned to show its original performance.

G1. G1GC (G1) is a highly-tunable partially-concurrent garbage collector which prioritizes latency over throughput. It is designed to replace CMS in the future versions of OpenJDK. We have manually tuned the value of *MaxGCPauseMillis* for performance consideration. Since G1 is an experimental collector in OpenJDK 8, we also try the later OpenJDK 9 for evaluation. However, OpenJDK 9 is not supported by the coupon service and Cassandra, and our evaluation on SpecJBB2015 shows similar results for those two versions. Therefore, we only report the result for OpenJDK 8.

Shenandoah. Shenandoah is a work-in-progress mostly-concurrent garbage collector. The application latency is quite low, but the introduction of *read barriers* and other concurrent phases strongly affect CPU utilization.



(a) low-throughput (b) high-throughput

Figure 9: The p99 latency results on SpecJBB2015 for different collectors with various throughput settings.

5.1 SpecJBB2015

We use the preset mode to evaluate SpecJBB2015 atop different collectors to understand its performance under various levels of throughput. In Alibaba, different throughput settings can be used to simulate different types of workload. The experiment is running on a physical machine with dual Intel Xeon Gold 6138 CPUs (80 logical cores) and 16GB Java heap size. The number of concurrent GC threads is set as the default value (53). We tune G1 with different *MaxGCPauseMillis* values and find that it reaches the shortest per-GC pause time when the value is 50ms, so we adopt this value for evaluation (referred to as *G1-tuned*). Figure 9 shows the 99th percentile latency for *Platinum*, CMS, G1, and Shenandoah. The results in Figure 9a show that *Platinum* always performs better than CMS under moderate throughput, and the 99th percentile latency is reduced by 38.4%-79.3%. *Platinum* also achieves comparable performance against our tuned G1. The improvement in latency mainly thanks to the mostly-concurrent collection in *Platinum*.

When the throughput becomes higher (shown in Figure 9b), G1 reaches its limit at 25000 requests per second. As a mostly-concurrent collector, the latency of Shenandoah is ultra-low in low throughput but dramatically rises when the throughput is 17000 and also reaches its limit at 25000. Note that G1 and Shenandoah shares a similar concurrent marking algorithm, and the main difference in their design choices is that G1 pauses mutators during collection while Shenandoah allows concurrent execution. Therefore, G1 has better GC efficiency and performs better under high throughput, while Shenandoah induces shorter pauses and performs better under low throughput. In contrast, *Platinum* can sustain the highest throughput of all collectors, thanks to the cost-effective isolated execution model during concurrent collection.

We also measure the CPU utilization for collectors under three different QPS settings (5000, 15000, 25000) to represent low, moderate, and high throughput. As Table 2 shows, the CPU utilization of *Platinum* is only slightly higher than CMS and better than both G1 and Shenandoah under all settings. Since we did not tune CMS for application latency, it reaches reasonable CPU consumption but far worse tail latency compared against other collectors. When the throughput reaches 25000, both G1 and Shenandoah show considerable CPU consumption compared with *Platinum*, which results in the severe tail latency problem (Figure 9b), while the CPU utilization in *Platinum* is still moderate.

Name	CMS	G1	Shenandoah	Platinum
Specjbb (qps=5000)	14.57%	16.53%	17.85%	15.11%
Specjbb (qps=15000)	31.77%	37.25%	43.03%	32.79%
Specjbb (qps=25000)	48.79%	77.66%	77.80%	50.56%
Cassandra (qps=80000, RI)	11.87%	14.35%	14.07%	12.99%
Cassandra (qps=80000, WI)	12.10%	15.97%	14.93%	13.79%
Coupon (qps=4000)	38.47%	36.17%	83.05%	34.50%

Table 2: The CPU utilization for four garbage collectors, with different applications

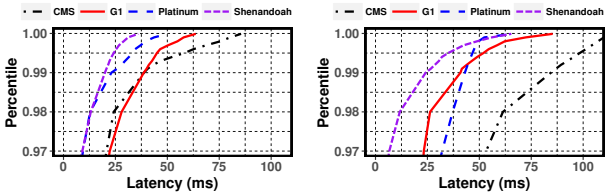
5.2 Cassandra

We evaluate Cassandra under the same settings as SpecJBB2015. Two different types of workload are leveraged for evaluation: (1) read-intensive workload (RI) with 76000 reads and 4000 updates per second; (2) write-intensive workload (WI) with 40000 reads and 40000 updates per second. We have also tuned G1 to achieve its best performance, and the value of *MaxGCPauseMillis* is 10ms. Figure 10 illustrates the tail latency for both scenarios with CDFs. As for the read-intensive workload, *Platinum* has comparable performance with Shenandoah, and improves the 99th percentile latency by 40.5% and 40.4% for CMS and our tuned G1 respectively. In *Platinum*, the latency for 97.2% of requests is less than 10ms, and the number is 9.5% and 3.2% larger than G1 and CMS. The improvement drops for write-intensive workload, and only 91.2% of requests finish in 10ms. This can be explained by more violated writes from mutators due

to more update operations on the globally-visible data structures. Nevertheless, the p99 latency of *Platinum* is comparable with our best-tuned G1 and improved by 44.9% compared with CMS.

All collectors show moderate CPU consumption in Cassandra. It is because Cassandra is an I/O-intensive application and spends much more time on accessing its storage compared with other scenarios. Since our tuned G1 reduces the application latency by greatly shrinking the young space and increasing the accumulated GC time, it reaches the highest CPU utilization among all collectors.

Table 3 further shows GC-related statistics in 30 seconds among different collectors in the read-intensive workload. It also shows the results for three different settings in G1. The untuned CMS has the least overall time among all collectors (except for G1-100ms and G1-60ms), but its average pause time is relatively large. As for G1, when setting *MaxGCPauseMillis* from 100 to 10, the overall GC time is enlarged by 2.5X, which results in higher CPU consumption. Compared with other collectors, Platinum reaches both satisfying average GC pause time (close to Shenandoah) and overall GC time (close to G1-100ms).



(a) Read-Intensive (b) Write-Intensive

Figure 10: The CDF results for Cassandra under two different workload

GC settings	Average pause (ms)	Overall time (ms)	p99 latency (ms)
CMS	28.168	366.189	38.776
G1-10ms	16.144	1037.864	38.677
G1-60ms	38.998	775.032	62.794
G1-100ms	58.739	413.583	76.732
Shenandoah	3.97	522.499	27.700
Platinum	4.66	433.335	23.061

Table 3: GC and latency statistics for Cassandra RI

5.3 Coupon

We finally show the performance of *Platinum* on the coupon service. Since we do not have enough physical MPK-enabled machines to conduct the clustered evaluation in Section 2.2, this evaluation still exploits the single-point environment mentioned in Section 2.3 on a 96-core machine with 16GB Java heap. The throughput is set to 4000 requests per second to simulate stressful throughput in the production environment. Since we have studied the performance of G1 on the coupon service before, the value of *MaxGCPauseMillis* is set to 60ms.

Figure 11 shows the CDF generated according to the response time of requests. The results indicate that *Platinum* can mitigate the long tail problem, especially for p99 latency. Thanks to the cost-efficient garbage collection, the p99 latency in *Platinum* is improved by 66.8% and 23.5% for CMS and G1. Since the real-world workload also contains requests whose response time is greatly extended by lags in other remote services, *Platinum* cannot help to improve them much and result in slightly better p999 latency against other collectors. The application latency for Shenandoah is very large (for example, the 99th percentile latency is 3.6s), which is out of range in Figure 11. Compared with statistics in Table 1, the average pause time and GC count in *Platinum* is 7.247ms and 162 respectively.

As for CPU utilization, *Platinum* consumes 34.5% of overall CPU resources, which is the smallest among all collectors. The CPU consumption is 1.67% and 3.97% smaller than G1 and CMS. As for Shenandoah, the CPU is close to saturated (83.05%), which can explain why application latency is quite large. To conclude, the evaluation results on all three applications confirm that *Platinum* finds a sweet spot between low application latency and moderate CPU utilization.

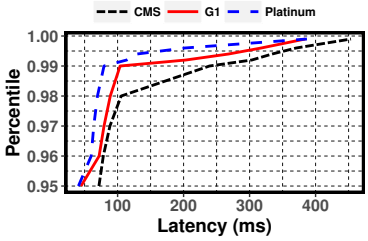


Figure 11: The CDF results for the coupon service

5.4 Breakdown analysis

Varying the pinned area size. Table 4 shows the runtime statistics of the Cassandra-WI workload with different sizes of the pinned area. When enlarging the pinned area, the average number of page faults for each GC cycle decreases, and the tail latency can be slightly improved. However, since the pinned area will only be reclaimed in the next GC cycle, enlarging its size will reduce the available memory in the eden space and increase the overall GC time. Therefore, users can tune the size of the pinned area to reduce the tail latency according to the application behavior.

Area size	Overall GC time (ms)	Avg. page faults	p99 latency (ms)
1/128	1007.327	13738	43.038
1/32	1106.951	12137	41.391
1/16	1145.982	11879	43.773
1/8	1206.859	10237	39.619

Table 4: GC-related statistics for Cassandra WI

GC Performance breakdown. This experiment breaks the accumulated GC time of Cassandra-WI into phases. As shown in Figure 12, mutators are allowed to run concurrently with GC threads most of the time (78.3%). Since the initial marking phase only scans the thread stacks, it lasts shortly and only takes up 1.1%. Meanwhile, the STW scavenge phase accounts for 15.6% to modify the stale references

in the pinned area and the collection area.

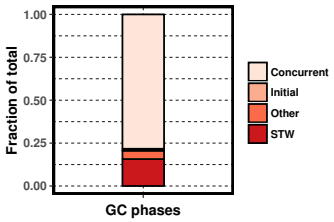


Figure 12: Phase-level breakdown for *Platinum*

The performance on Spark. We also evaluate the performance of *Platinum* on Spark, with its built-in *PageRank* application. Our evaluation shows that the execution time with *Platinum* is 7.93% longer compared with G1. Since the memory and GC behavior in Spark is much different from interactive services, it induces more page faults and larger stop-the-world pauses, which is the main reason for performance slowdown.

6 Related Work

Reducing pause time. STW pauses introduced by garbage collections have been studied for years. For STW garbage collectors, some work aims to reduce the pause time by sufficiently leveraging the computing resources. Gidra et al. [15, 16, 17] study the performance of PSGC in NUMA machines and provide NUMA-aware optimizations. Suo et al. [40] and Qian et al. [37] refine the work-stealing algorithm to offer a more scalable minor GC algorithm. Another line of work focuses on designing new concurrent collectors to co-run mutators with GC threads to reduce the pauses. In the OpenJDK HotSpot JVM, the latency-aware G1GC [11] has recently become the default collector, and two mostly-concurrent collectors, ZGC [33] and Shenandoah [13], are also attractive. Stopless [36] provides real-time GC support while preserving lock-freedom and fragmentation control. Österlund et al. [34] improve the pause time of G1GC with a non-blocking handshake between threads. The design of *Platinum* learns from those concurrent collectors to propose a CPU-efficient solution.

The intensive usage of language runtime in the big data area has stimulated studies on building big-data-friendly garbage collector with low pauses. Nguyen et al. [31, 32] put the data objects generated by the big data systems into segregated spaces where objects are managed with separated GC algorithms. Gog et al. [18] provide a similar region-based algorithm but mainly for the CLR runtime. Bruno et al. [6, 7, 8] divide the heap into many generations and pre-tenure objects that are believed to live long through runtime analysis. *Platinum* is built for reducing pauses for another kind of scenario: latency-sensitive, session-based interactive services.

Hardware-assisted GC. Hardware features also affect the design choices of garbage collectors. Prior work has studied on improving the performance of GC with primitives available in commodity hardware. Belay et al. [3] leverage virtualization technology to escalate Java runtime to the non-root ring 0 mode and accelerate GC with VM page management. Ugawa et al. [42] enhance the original Sapphire garbage collector with the RTM feature, and Ritson et al. [39] explore the usage of RTM in various collectors. Wu et al. [45] extend the area of garbage collector from normal DRAM to non-volatile memory (NVM) and studies crash consistency issues during GC. *Platinum* leverages RTM and MPK features to build a new concurrent garbage collector.

Except for commodity hardware, there is also a trend to build customized hardware, or GC accelerators, for various considerations. Azul Systems has built a customized system including CPU, chip, board, and OS to run garbage collected JVMs efficiently. Their GC algorithm is also largely modified to leverage those customized features [10, 23, 41]. Mass et al. [26] build a hardware GC accelerator to achieves higher GC throughput and lower power consumption.

Runtime optimization in distributed environments. Distributed applications are running atop multiple runtimes on different machines. Maas et al. [25, 27] show that GC in a single JVM can have a magnified effect on the whole applications and proposes policies to orchestrate GC among different JVMs. Lion et al. [24] find frequent JVM re-start in task-based workload and provides a JVM pool to skip the time-consuming warm-up phase. Nguyen et al. [30] optimize the communication between JVMs by providing an efficient serialization protocol, while Navasca et al. [29] allow Java threads to directly operate on the serialized byte streams with compiler techniques. Wang et al. [44] propose to dynamically change the memory limits of JVMs to fit the cloud environment. Both Fang et al. [12] and Călin et al. [22] leverage efficient spilling to reduce the memory footprint of large data-parallel applications. *Platinum* also quantifies the effect of GC in a distributed cloud environment and proposes a hardware-assisted algorithm to optimize the GC performance.

7 Conclusion

Latency and CPU efficiency are both essential for interactive services. Unfortunately, traditional garbage collectors cannot achieve both goals at the same time. This paper provides *Platinum*, a collector allowing concurrent but isolated execution of mutators and GC threads, with hardware assistance. The evaluation shows that *Platinum* significantly reduces the tail latency while preserving moderate CPU utilization compared with prior concurrent garbage collectors.

8 Acknowledgement

We sincerely thank our shepherd Gilles Muller and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61672345, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Haibo Chen is the corresponding author.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2009.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Notices*, volume 23, pages 11–20. ACM, 1988.

[3] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

[4] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI*, volume 91, pages 157–164. Citeseer, 1991.

[5] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262. ACM, 1984.

[6] R. Bruno and P. Ferreira. Polm2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 147–160. ACM, 2017.

[7] R. Bruno, L. P. Oliveira, and P. Ferreira. Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications. *ACM SIGPLAN Notices*, 52(9):2–13, 2017.

[8] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 28. ACM, 2019.

[9] A. Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, page 13, 2014.

[10] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56. ACM, 2005.

[11] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[12] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 394–409. ACM, 2015.

[13] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 13. ACM, 2016.

[14] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. {IMIX}: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018.

[15] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 7. ACM, 2011.

[16] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *ACM SIGPLAN Notices*, volume 48, pages 229–240. ACM, 2013.

[17] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: a garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[18] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[19] S. Han, S. Lee, S. S. Hahn, and J. Kim. Syncgc: A synchronized garbage collection technique for reducing tail latency in cassandra. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, page 20. ACM, 2018.

[20] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[21] R. L. Hudson and J. E. B. Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57. ACM, 2001.

[22] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 97–109, 2017.

[23] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The collie: a wait-free compacting collector. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[24] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 383–400. USENIX Association, 2016.

[25] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *ACM SIGOPS Operating Systems Review*, 50(2):457–471, 2016.

[26] M. Maas, K. Asanović, and J. Kubiatowicz. A hardware accelerator for tracing garbage collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 138–151. IEEE Press, 2018.

[27] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[28] S. Microystems. Memory management in the java hotspot™ virtual machine, 2006.

[29] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 538–553. ACM, 2019.

[30] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ACM SIGPLAN Notices*, volume 53, pages 56–69. ACM, 2018.

[31] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[32] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, 2015.

[33] OpenJDK. ZGC - The Z Garbage Collector. https://openjdk.java.net/projects/zgc/, 2019.

[34] E. Österlund and W. Löwe. Block-free concurrent gc: stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 1–12. ACM, 2016.

[35] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[36] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pages 159–172, 2007.

[37] J. Qian, W. Srisa-an, D. Li, H. Jiang, S. Seth, and Y. Yang. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for java vm. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 170–181. ACM, 2015.

[38] J. Qian, W. Srisa-An, S. Seth, H. Jiang, D. Li, and P. Yi. Exploiting fifo scheduler to improve parallel garbage collection performance. *ACM SIGPLAN Notices*, 51(7):109–121, 2016.

[39] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with haswell hardware transactional memory. In *ACM SIGPLAN Notices*, volume 49, pages 105–115. ACM, 2014.

[40] K. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*, page 35. ACM, 2018.

[41] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46(11):79–88, 2011.

[42] T. Ugawa, C. G. Ritson, and R. E. Jones. Transactional sapphire: Lessons in high-performance, on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(4):15, 2018.

[43] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1221–1238, 2019.

[44] J. Wang and M. Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 745–758, Santa Clara, CA, 2017. USENIX Association.

[45] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 70–83. ACM, 2018.