A Unifying Approach to Supporting Multiple Garbage Collectors in AOT-compiled Binaries

Thomas Schrott
Johannes Kepler University
Linz, Austria
thomas.schrott@jku.at

Christian Häubl
Oracle
Linz, Austria
christian.haeubl@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Abstract

Some language implementations combine garbage collection with ahead-of-time compilation to produce self-contained executables for managed-language programs. In these systems, one can typically choose a garbage collector (GC) only at build time. To use another GC, e.g., for better performance, one needs to build another executable.

In this paper, we present an approach for supporting multiple GCs in the same self-contained executable using unified barriers, object layout, object header, and dynamic dispatch. This enables developers to select a GC at run time. Additionally, isolates, i.e., lightweight virtual machine instances with separate collected heaps but within the same process, can now use different GCs alongside each other.

We evaluate our approach in GraalVM Native Image, supporting the Garbage First (G1) and the Serial GC in the same executable. Our evaluation on the DaCapo Chopin and Renaissance benchmarks shows that G1 has on average no performance change (min. −9 %, max. 14 %). Serial GC shows a peak performance regression of 11 % (min. −10 %, max. 33 %). We believe the simplicity of the approach and that one can now choose the GC at run time and on a per isolate basis make this overhead acceptable.

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

Keywords: Garbage Collection, Ahead-of-Time Compilation, GraalVM Native Image, Write Barrier, G1

ACM Reference Format:
Thomas Schrott, Christian Häubl, Hanspeter Mössenböck, and Stefan Marr. 2026. A Unifying Approach to Supporting Multiple Garbage Collectors in AOT-compiled Binaries. In *Proceedings of 23nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26)*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

MPLR '26, Brussels, Belgium

1 Introduction

Garbage collectors (GCs) are widely used by many language implementations for automatic memory management. However, different GC algorithms each have their tradeoffs in terms of throughput, memory footprint, and latency [9], and no single GC is best for every application [9, 29].

For language implementations using ahead-of-time (AOT) compilation for producing self-contained native executables, this is a challenge. Since GCs differ widely in their needs for barriers, object layout, and bits in the object header, AOT-compiling language implementations optimize for performance and require a GC to be chosen at build time. Thus, the resulting executable contains exactly one GC. To use a different GC, one has to build and ship another executable.

One such language implementation is GraalVM Native Image [13, 31], which AOT-compiles Java bytecode to self-contained native executables. Unlike other Java implementations, it supports isolates, which are lightweight virtual machine (VM) instances within the same OS process. However, since the GC has to be selected at build time, all isolates must use the same GC. Native Image's Garbage First (G1) implementation is directly derived from HotSpot's [14, 20] to avoid having to maintain multiple versions. This also means that G1 supports only a single isolate. Thus, if multiple isolates are to be used, G1 is not an option.

To improve the flexibility of Native Image, we look for a mechanism that avoids adding unmanageable complexity to the GC implementations, allows us to continue to use the HotSpot-derived G1, and allows us to select the GC at run time on a per isolate basis. At the same time, we want to avoid the need to have GC-specific code versions or perform code patching, as this would lead to a larger binary size or to higher memory usage at run time. Thus, we want to use the same AOT-compiled code for all GCs. While this will not lift G1's restriction of being limited to one isolate, it allows us to

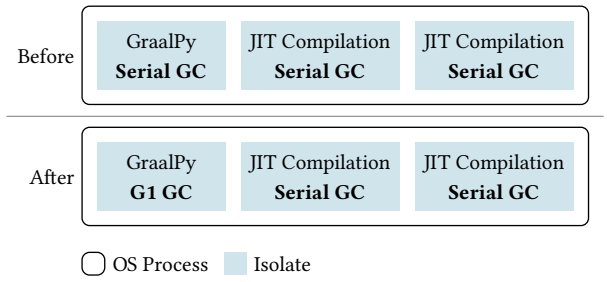

Figure 1. With only one GC in the executable, all isolates have to use the Serial GC. After including G1 and the Serial GC, the GraalPy-isolate can use G1 while JIT-compilation-isolates still use the Serial GC.

use G1 for the performance-critical isolate and use the Serial GC for the other ones. Serial GC is a simple, non-parallel, non-concurrent, generational stop-and-copy GC [14].

One concrete use case for us are Truffle-based [22, 23, 32] language implementations such as GraalJS [16], GraalPy [17], or GraalWasm [18]. Truffle is a language implementation framework for creating interpreters. It uses Native Image to AOT-compile the interpreter. At run time, it uses separate isolates for just-in-time (JIT) compilation to create high-performance code for the hot parts of the executed user program. Figure 1 shows an example of the isolates used by GraalPy. The main, performance-critical GraalPy-isolate executes the interpreter and the JIT-compiled Python code. The JIT compilation isolates compile the hot parts of the Python code. Normally, the executable contains only a single GC, so all isolates have to use the Serial GC as shown in the before part of Figure 1. Including multiple GCs in the same executable, allows the GraalPy-isolate to use and benefit from G1 while the JIT compilation isolates still use the Serial GC, as illustrate in the after part of the figure.

An optimal language implementation would specialize the AOT-compiled code with GC-specific details, for instance, by inlining barriers, adapting the object layout, and object header to accommodate the necessary bits for the GC [9, 28] and achieve best possible performance.

However, achieving the maximal possible specialization for multiple GCs within the same OS process is not possible without increasing executable size or run-time memory usage significantly, which is often a critical concern for deployment in production. Thus, we are willing to accept minor performance drawbacks. Therefore we chose to unify the GC-specific parts and use them with all included GCs. For the interface between the runtime system and the GC, we use dynamic dispatch on slow path to delegate to the currently used GC. This allows us to use the same specialized AOT-compiled code for all GCs while keeping the impact on binary size and memory usage acceptable.

We implemented this approach in GraalVM Native Image adding support for G1 and Serial GC in the same AOT-compiled executable. We evaluate it using subsets of the DaCapo Chopin [1] and Renaissance [25] benchmark suites that are supported by Native Image.

Comparing a G1-only executable to an executable containing both GCs that uses G1 at run time shows on average no change for the performance (min. −9 %, max. 14 %). The maximum resident set size (RSS) when using G1 remains on average unchanged as well (min. −11 %, max. 6 %). Comparing a Serial GC-only executable to an executable containing both GCs that uses the Serial GC at run time shows on average an 11 % performance regression (min. −10 %, max. 33 %) and a 22 % max. RSS regression (min. −29 %, max. 84 %). The executables with both GCs are on average 3 % larger than ones containing only G1, and 21 % larger than ones containing only the Serial GC. Furthermore, we found that the simple dynamic dispatch for slow path calls does not introduce a noticeable performance overhead.

Limitations of our approach are that all GCs need to use the same object layout and that they need to have similar barriers. Otherwise, the performance impact can outweigh the benefits of our simple approach.

Our main contributions are:

- an approach to support multiple GCs in a single AOT-compiled executable using unified barriers, object layout, object header, and dynamic dispatch (Section 3)
- an implementation in GraalVM Native Image supporting both the G1 and the Serial GC in a single executable (Section 3.2)
- an evaluation on the DaCapo Chopin and Renaissance benchmarks (Section 4)

2 Background: GraalVM Native Image

GraalVM Native Image [13, 15, 31] AOT-compiles Java bytecode to an optimized native executable. We briefly discuss it and its technical details relevant for this paper.

Compared to a standard Java Virtual Machine (JVM), Native Image provides several advantages. It contains only the reachable code and classes that are used. Its native executables start up faster, since all code is compiled and some components are initialized when creating the executable. Thus, they do not require the warmup phase of a JIT-compiled system. The native executables are self-contained and include the SubstrateVM with its runtime system and GC.

2.1 Build-Process

Figure 2 shows the Native Image build process [15, 31]. The `native-image` builder takes Java bytecode as input. This includes used libraries, the Java Development Kit (JDK), the application classes and the SubstrateVM.

This paper is based on Schrott's master's thesis [26].

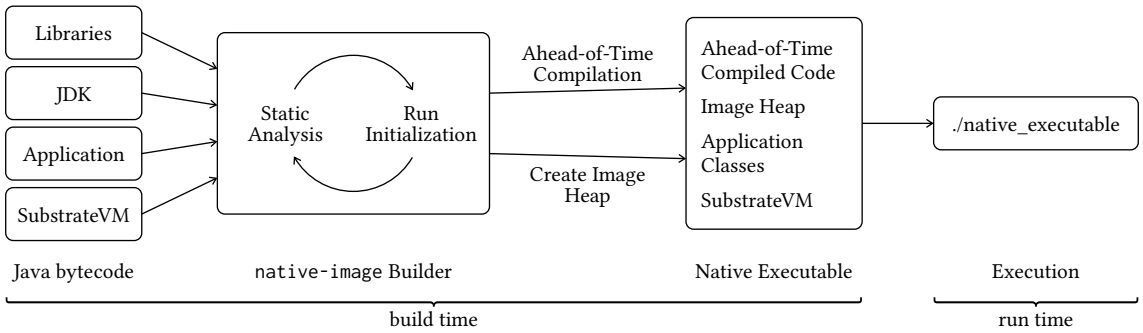

Figure 2. Native Image build process (based on Wimmer et al. [31])

Then, the builder uses static analysis with a closed-world assumption to mark all reachable elements that are used at run time to include them in the executable. Currently, it is not possible to load additional classes at run time. Additionally, the `native-image` builder initializes the relevant SubstrateVM and user classes. Initialization code can allocate Java heap objects. If they are needed at run time, they are stored in the image heap, a special area of the native executable. Once the builder reaches a global fixed point, it compiles all reachable bytecode to machine code and creates the image heap. The created executable is then specific for a certain operating system and processor architecture.

2.2 Heap Structure and Isolates

At run time, the Java heap is split into two parts [31]. The *image heap* contains all reachable Java objects that were already allocated at build time. The *collected heap* contains all Java objects that are allocated at run time and only this part is garbage collected.

An isolate is a lightweight VM instance within the same OS process [30, 31]. Each isolate has its own Java heap. The image heap is made available as a copy-on-write mapping to ensure isolation between isolates. Generally, an isolate cannot access the Java heap of other isolates and references across isolate boundaries are not possible. This means that each isolate can run the GC independently.

Figure 3 shows a process with two isolates. The AOT-compiled code is mapped only once and shared between all isolates. At run time, each isolate reserves a dedicated address space for its Java heap. The image heap is memory mapped to the start of this address space (the heap base). Since Java heap accesses are relative to the heap base, isolates share unmodified parts of the image heap to conserve physical memory. The individual and collected heap of each isolate starts after the image heap.

2.3 Supported GCs

Native Image supports three different GCs: Serial, Epsilon, and G1 GC [14]. A GC is selected at image build-time and

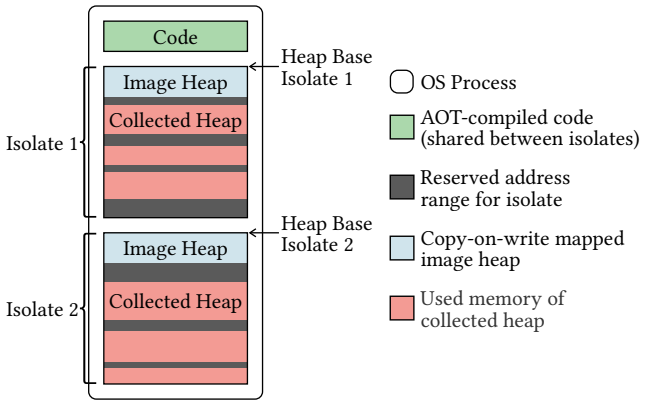

Figure 3. Example application using two isolates. (based on Wimmer [30])

only that GC is included in the created executable. If one wants to use another GC, one has to build another executable.

The Epsilon GC is a no-op GC that does not collect garbage and does not use any barriers [14]. We therefore focus only on supporting G1 and Serial GC in the same executable as it is the more relevant use case.

2.3.1 Conceptual Overview. The collected heap is a generational heap and split into a young and an old generation. Each generation is further split into smaller blocks.

When a GC collects only parts of the heap, e.g., only the young generation, it has to consider references that are in other parts, and point into the collected part, as additional GC roots. We refer to these references as *interesting references*. Native Image tracks these interesting references with card-table-based remembered sets, where each card corresponds to 512 bytes of Java heap memory [9, 34].

A dirty card means, that the corresponding heap memory contains an interesting reference, for instance, a cross-generation reference. A clean card means that the corresponding heap memory does not have such a reference. For performance, each card uses one byte. When collecting only parts of the heap, the GC scans the card table for dirty cards

and their corresponding heap memory for interesting references [7, 9]. For visiting all such references, the GC has to iterate over all objects of dirty cards. The first object of a card however may start in an arbitrary previous card. The GC uses a second table, the first object table, that stores where the first object of a card starts.

Conceptually, the write barrier is a small piece of code that is responsible for adding a reference to the remembered set. For card-table-based remembered sets, the write barrier dirties the card. The implementation details of this differ between GCs. Native Image uses two types of write barriers. *Precise write barriers* dirty the card corresponding to the address where the interesting reference is stored into. Here, the GC only has to visit the references within the dirty card. *Imprecise write barriers* dirty the card corresponding to the start of the object the reference is stored into. Thus, the GC has to visit all references of an object, even though they may be in different cards.

Additionally, in Native Image a write barrier can be *filtering*. When it is filtering, only interesting reference writes lead to a dirty card [34]. This reduces the number of dirty cards the GC has to visit.

2.3.2 Serial GC. The Serial GC is a simple, non-parallel, non-concurrent, generational stop-and-copy GC [14]. It is the default GC for Native Image. Each generation contains two semispaces [2, 5, 9].

Each semispace is divided into *chunks*. It distinguishes between *aligned* and *unaligned chunks*. Aligned chunks have a fixed size that is set at build time. The default size is 512 KB and an aligned chunk can contain multiple objects. The chunk start is aligned to the chunk size, hence the name. In contrast, unaligned chunks contain exactly one object and the chunk size depends on the object size. The start of an unaligned chunk is not aligned and multiple consecutive unaligned chunks do not have padding between them. They are used for objects larger than a certain threshold.

The Serial GC uses imprecise write barriers for all objects. For the Serial GC, references from old generation objects to young generation objects are interesting references, which lead to dirty cards. Additionally, the card table and the first object table are directly stored at the start of each chunk.

2.3.3 G1 GC. The G1 GC is a generational, parallel, mostly concurrent stop-the-world evacuating GC [3, 14]. It achieves short pause times and low latency while keeping a high throughput and thus, is better suited for larger heaps than the Serial GC. G1 is based on HotSpot's implementation, written in C++, and uses process-global state, which means it supports only one isolate. Changing G1 to avoid using global state and be isolate aware, would make its implementation harder to maintain and update. So, if one wants to use more than one isolate, they currently must all use the Serial GC.

G1 splits its generations into *regions*. The region size has to be selected at build time and defaults to 1 MB, which is also the minimum region size. It uses different write barriers depending on the object type. While arrays use precise write barriers, all other objects use imprecise write barriers. For G1 all region-crossing references are interesting ones, as it may not collect all regions of a generation at once. In contrast to the Serial GC, G1 stores the card table and the first object table separate from its regions in large contiguous blocks of memory. Additionally, G1 uses snapshot-at-the-beginning (SATB) barriers for concurrent marking [3, 35]. During a GC, the SATB barrier records the old value of a reference before it gets overwritten. This ensures all objects that were live at the beginning of the collection stay live.

3 Unified Barriers and Dynamic Dispatch

To achieve the goal of a single executable supporting runtime-selectable GCs, we propose to use an approach that unifies the GC-specific parts as much as possible, most importantly, the GC barrier code inlined into the application code, the object layout, and the object header use. For parts that are not performance-critical and differ between GCs, we dispatch to the relevant implementation dynamically at run time. This approach enables us to use different GCs within the same OS process without having to produce separate code versions or performing code patching as this would lead to a larger binary size or to higher memory usage.

First, we discuss the key ideas of our approach before detailing the implementation.

3.1 Key Ideas

Unified Barriers. Replacing the GC-specific barriers with a unified implementation must not impact the correctness. Thus, the unified barrier must still ensure all invariants of the GC-specific ones. For example, the unified write barrier must dirty at least the cards that the GC-specific write barrier dirtied. Of course, unifying GC-specific implementations that were developed separately and not meant for this use case introduces some challenges. The concrete challenges are specific to the GCs and one cannot really generalize them. Still, we think our general approach is simple and works well, but if one applies it to an existing system, there are some issues one can expect. The challenges we have faced for our implementation are described in Section 3.2.3.

Object Layout and Object Header. The object layout is typically GC-specific and determines the object alignment, the location of object fields, and the reference size. Since this affects the machine code, it needs to be unified for all GCs.

The object header contains information used by the runtime system such as the class pointer. Since this is independent of the GC, we need to ensure that it is uniform. However, the header also contains GC-specific bits. Since in Native Image objects are local to an isolate and only accessible from with in it, we only need to ensure that the same number of bits are reserved. Their specific use can remain GC-specific.

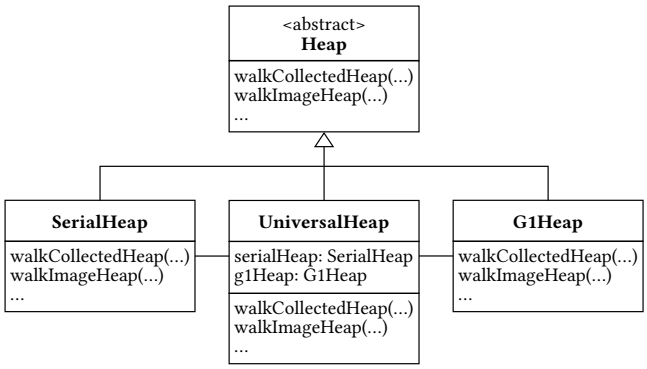

Figure 4. Architecture for including multiple GCs. `UniversalHeap` implements the dynamic dispatch to the GC-specific implementations

Collected Heap. The collected Java heap is created at run time. Therefore, it is only used by a single GC and it can remain GC-specific. Thus the only requirement is, that its structure, e.g., the layout and location of the generations, has to be compatible with inlined GC-specific code, such as the unified barriers.

Dynamic Dispatching. The VM uses a common interface for interacting with the GC [19, 20]. When the executable contains only a single GC, these interface methods can be statically linked. When including multiple GCs, we propose using dynamic dispatching for executing the correct GC-specific implementation. None of these calls are on the fast paths. Otherwise, this would create performance issues.

3.2 Implementation

We have implemented this approach in GraalVM Native Image. We focused on the combination of the G1 and the Serial GC. So, at build time both GCs need to be included in the created executable. Later at run time, users can select which GC to use on a per isolate basis.

For unifying the barriers and the object layout, we adapted the Serial GC to use the G1-specific implementations as the Serial GC is the simpler of the two GCs.

3.2.1 Dynamic Dispatching. SubstrateVM uses abstract classes and Java interfaces for defining a common GC interface. Each GC implements this interface to provide its specific behavior. For example, Figure 4 shows the UML-diagram for the `Heap` class. SubstrateVM calls only the methods defined in the abstract class. `SerialHeap` overrides these methods with the implementation of the Serial GC and `G1Heap` with the implementation of G1.

When building an image that contains only a single GC, only one of these GC-specific subclasses is initialized and stored as a singleton. SubstrateVM uses this singleton for calling the common interface methods. The static analysis notices that the implementation of the other GC is unused and automatically excludes it from the created executable.

Listing 1. Dynamic dispatching to GC-specific implementation using an isolate-local flag.

```
1 @Override
2 public boolean walkCollectedHeap(
3     ObjectVisitor v) {
4   if (SubstrateOptions.useSerialGC()) {
5     return serialHeap.walkCollectedHeap(v);
6   }
7   return g1Heap.walkCollectedHeap(v);
8 }
```

Listing 2. Use same implementation for all GCs.

```
1 @Override
2 public boolean walkImageHeap(
3     ObjectVisitor visitor) {
4   return g1Heap.walkImageHeap(visitor);
5 }
```

For including multiple GCs in the same executable, we added a third implementation for all abstract classes and interfaces. Each class has fields for storing references to the existing GC-specific implementations. Figure 4 also shows the new `UniversalHeap` class. Its methods simply forward the method calls to one of the existing implementations, based on the GC used by the current isolate. For implementing the dynamic dispatch, we use an isolate-local flag, that specifies the GC of an isolate. Most methods check this flag and forward the method call to the implementation of the used GC. As an example for this, Listing 1 shows the code of `walkCollectedHeap`.

Some methods must be identical for both GCs, e.g., the ones regarding the object layout or the image heap. These methods always forward the method call to the same implementation regardless of the used GC. `walkImageHeap` is such a method. It always calls the G1 implementation. Listing 2 shows its source code.

During the image build, we instantiate all three implementations. But we register only the third new implementation, e.g., `UniversalHeap`, as the singleton. The GC-specific instances are stored in their respective field of the new classes. Now, the static analysis marks all three implementations as used and automatically includes them in the executable. Additionally, all usages of the singletons can remain unchanged.

Adding a third implementation, like `UniversalHeap`, and forwarding method calls is sufficient for most GC-specific classes. Some parts such as the object layout, the object header and the write barriers need special handling.

Most Java heap objects are allocated in a thread-local allocation buffer using a simple bump-pointer allocation. This mechanism was already GC-independent. The allocation

slow path is GC-specific and uses dynamic dispatching to call the code of the current GC.

3.2.2 Object Layout and Object Header. Using the same machine code for both GCs requires the runtime to use the same object layout and compatible object headers. The object layout is already a common class for all GCs. But the different GCs initialize it with different GC-specific values. So, an additional implementation is not necessary, but G1 and the Serial GC need to use the same layout. Again, we adapted the Serial GC to use G1's object layout, which includes a mark word in the object header. Adding the mark word increases the size of the object header for the Serial GC.

The object header contains the Java class pointer, a mark word and some reserved bits used by the GCs. It is important that the layout of the object header is identical for both GCs. This means, the Java class pointer must be at the same position and the number of reserved bits must be the same. How the GCs use the reserved bits can remain GC-specific. For example, the Serial GC uses the reserved bits for storing the chunk type, i.e., whether the object is stored in an aligned or an unaligned chunk, and for storing whether an object is old and reference writes should be recorded in the remembered set [19]. G1 uses the reserved bits as a mark bit and as age bits [20]. To share the initial image heap between both GCs, image heap objects must initially use the same bit pattern for these bits. Coincidentally, both GCs already use the same initial pattern, so no additionally change was necessary.

3.2.3 Barriers. The biggest challenge for including both the G1 and the Serial GC in the same executable are the barriers, as they are mostly inlined into the AOT-compiled code. G1 uses SATB barriers and more complex write barriers that perform precise or imprecise card marking depending on the object type. Additionally, G1 uses read barriers in some cases, e.g., for weak references. The Serial GC uses simpler and only imprecise write barriers.

The fast path of G1's SATB barrier checks if concurrent marking is enabled. If concurrent marking is active, the slow path adds the object to the SATB queue. If the queue is full, the GC processes all objects. As the Serial GC does not need SATB barriers, we made sure that the SATB barrier always regards the concurrent marking as disabled. This way, the Serial GC never performs any slow path calls and does not need a SATB queue. The read barrier used by G1 is the same SATB barrier. So, no additional changes are necessary.

Unifying the write barrier was more challenging. First, G1 and the Serial GC use a different structure for their card-table-based remembered sets. The Serial GC stores the card table and the first object table at the start of each chunk. On the other hand, G1 uses a contiguous card table and first object table for the whole Java heap that is stored separate from its regions. So, the GC-specific write barriers calculate the card index differently. We changed the Serial GC to also store the card table and first object table in separate memory as G1 does, instead of as part of its chunks.

As G1 uses precise write barriers for arrays and imprecise ones for all other objects, we added support for precise write barriers to the Serial GC.

Finally, the write barriers of the two GCs perform different filtering, i.e., dirty the cards in different cases. The Serial GC write barrier only checks the bit in the object header whether an object is old. As only storing a young object in an old object needs to dirty the card table. The G1 write barrier instead checks if a written reference crosses a region boundary and writes within the same region do not dirty the card table. For correctness, the unified write barrier must dirty at least the same cards as the GC-specific barriers do. Thus, the filtering must not eliminate dirtying the card table for cross-generational reference writes. This imposes additional requirements on the collected heap of the Serial GC. Normally, the Serial GC uses 512 KB for its aligned chunks. G1 uses 1 MB for its regions. Due to this mismatch between chunk and region size, two aligned chunks take the same space as one G1 region. However, these two aligned chunks may belong to different generations and a reference write between them has to dirty the card. Unfortunately, the filtering G1 write barrier only checks for regions and since both chunks fit in the same region, it would not dirty the card table. We chose to avoid this problem by making the aligned chunks as large as G1 regions.

Another problem with this filtering occurs for unaligned chunks. Multiple consecutive unaligned chunks do not have any padding between them because they are not aligned (see Section 2.3.2). Again, cross-generation references that would be within the same G1 region are possible for which the G1 write barrier would not dirty the card table. To solve this problem, we align unaligned chunks so that their start is aligned to the G1 region size.

4 Evaluation

We evaluate our implementation of integrating the G1 and the Serial GC into the same GraalVM Native Image in terms of peak performance, overall memory use, and binary size.

4.1 Methodology

We used the DaCapo 23.11-MR2-chopin [1] and Renaissance 0.16.0 [25] benchmark suites. We used subsets of these benchmark suites that are supported by Native Image.[1]

For the best peak performance, we executed all benchmarks with profile-guided optimizations (PGO). We built the benchmarks once for each GC and executed each benchmark

[1]Native Image's closed world assumption results in some benchmarks not being compatible. From DaCapo, we exclude avrora, batik, biojava, cassandra, eclipse, graphchi, h2o, jme, jython, kafka, spring, tomcat, tradebeans, tradesoap, and zxing. From Renaissance, we exclude als, chi-square, db-shootout, dec-tree, gauss-mix, log-regression, mnemonics, movie-lens, naive-bayes, neo4j-analytics, and page-rank.

10 times. For each execution, we perform several warm-up runs followed by the one run that collects the results. The concrete number of warm-up runs depends on the benchmark, and ranges from 9 to 89. Based on the 10 data points, we calculated the median for the execution time and the max. RSS.[2] For benchmarking our multi-GC implementation, we used one executable that contains both GCs and selected a GC at run time. As baseline, we use an executable containing only one of the GCs and compare it to the executable containing both GCs.

We ran the benchmarks on an Intel Core i7-1355U with 2 performance cores, 8 efficiency cores, and 32 GB of memory. We set the maximum Java heap size to 8 GB as we wanted to measure the peak performance. To reduce measurement noise, we pinned the execution to the 8 efficiency cores using `hwloc-bind` and locked them to 2.1 GHz. We also ran the benchmarks on the performance cores but since the results were virtually the same, we did not include them.

4.2 Profile-Guided Optimizations

Oracle GraalVM supports PGO [21]. Using PGO requires a two stage build process for each benchmark. First, an instrumented image is built. During the so-called instrument run it collects the profiling data, which includes how often certain events occurred, e.g., how often a method was called or how often a branch was taken. This collected profiling data is then used to build an optimized image that is used for the benchmark runs.

During an instrumented run only one GC is executed. So, the profiling data is only collected for that GC. When there are multiple GCs in the executable, we do a separate profiling run with each GC and merge the profiles. Merging the profiles adds the counts from the individual profiles, which can affect optimizations as some optimizations use absolute thresholds for deciding whether to apply them. To achieve similar optimization decisions also for the baseline runs, we merged the profile with itself. We use separate profiles for each of the possible combinations of a specific benchmark and the selection of GCs included in the executable to evaluate the best-case scenario. We assume this represents Native Image's use in production, where one optimizes a specific application to achieve the best possible performance.

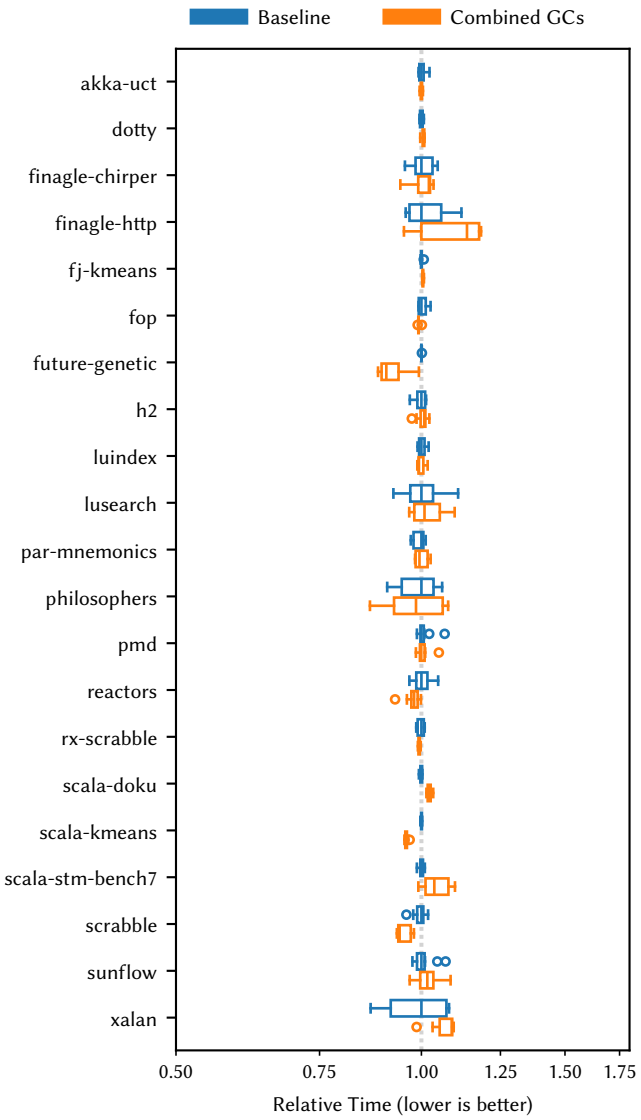

Figure 5. Execution times with G1 selected in the combined executable. As baseline, we use an executable containing only G1. On average and as we would expect, performance does not change (min. −9 %, max. 14 %).

4.3 Execution Time

As the peak performance metric, we use the execution time as reported by the individual benchmarks.

For G1, the geometric mean of the medians of the individual benchmarks shows no change in peak performance. Figure 5 shows the individual benchmark results with G1 as the baseline and G1 selected at run time in our combined version. *future-genetic* shows an improvement of 9 %. *finagle-http* has the worst regression of 14 %. The remaining benchmarks are somewhere in between. Some show a small regression and others a small improvement.

Since we did not change the implementation of G1 and still use the G1 write barriers and object layout, we also expected the performance to remain unchanged. We only added the run-time dispatch for slow-path calls to dispatch to the selected GC, which has no significant impact. We believe any performance difference in the measurements is caused by normal variations for instance by G1 deciding at different points in time to collect.

[2] Maximum RSS is the largest amount of physical memory an application used during its execution.

In contrast, we modified Serial GC to work with the G1 components. Most notably, we use a larger object header and G1's larger and more complex write barriers that add additional checks compared to the ones of the Serial GC. Thus, we expect reduced performance. Figure 6 shows the performance results with the Serial GC as the baseline and the Serial GC selected at run time in our combined version. The geometric mean of the medians shows an 11 % performance regression. *rx-scrabble* has the worst regressions with 33 %. But there are also benchmarks where performance improves, e.g., *fj-kmeans* improves by 10 %.

In our investigation we found the improvements with Serial GC to be caused by the use of G1's precise barriers for arrays. This means, the GC does not have to visit all objects referenced from arrays anymore, which is for some benchmarks better than Serial GC's imprecise barriers.

4.4 Overall Memory Use

In addition to performance, we also assess the memory use based on the max. RSS. Figure 7 shows the max. RSS results with G1 as the baseline and G1 selected at run time in our combined version. The geometric mean of all medians shows no performance change as we hoped, since the implementation is mostly unchanged. While there are some variations, most benchmarks show only small changes. However, the largest improvement shows *lusearch* with 11 % and the largest regression with 6 % is *reactors*.

Figure 8 shows the max. RSS results with the Serial GC as the baseline and the Serial GC selected at run time in our combined version. The geometric mean of all medians shows an overall regression of 22 %. *akka-uct* sees the largest memory reduction of 29 %. *lusearch* has the largest increase of 84 %. With the larger object header and doubling the size of aligned chunks, the results are expected.

Using a larger chunk size affects the GC policy. Specifically, the generations are sized based on the chunk size. Additionally, the thread-local allocation buffer uses a full chunk, making it larger than in the unmodified Serial GC. This can lead to later GCs and higher memory usage.

4.5 Binary Size

When including both G1 and the Serial GC in the same executable, we expect an increase in binary size. Figure 9 shows the relative binary size of executables compared to executables containing only G1.

The geometric mean shows on average a 3 % size increase for executables containing both the G1 and the Serial GC. *finagle-http* has the smallest increase of 1 %. *scala-kmeans* has the largest increase of 7 %. These size changes indicate the code increase caused by adding the Serial GC, which is ≈300 KB large. We found that some of the cost can be explained by changes to the encoding of thread-local accesses, which now take four instead of one byte offsets to access data relative to the thread-base register. Both, G1 and the Serial GC use separate thread-local state, which together go beyond the offsets that can be encoded in a byte and the resulting x86-64 machine code ends up being larger.

As the Serial GC uses smaller write barriers and the GC-code itself is smaller, all executables with only the Serial GC are smaller than the ones with only G1. When assessing the size difference with having only the Serial GC as the baseline, we find executables with both GCs are considerably larger. The geometric mean is 21 % larger. The smallest size difference has *h2*, which is only 17 % larger, while *par-mnemonics* has the largest difference with 26 %.

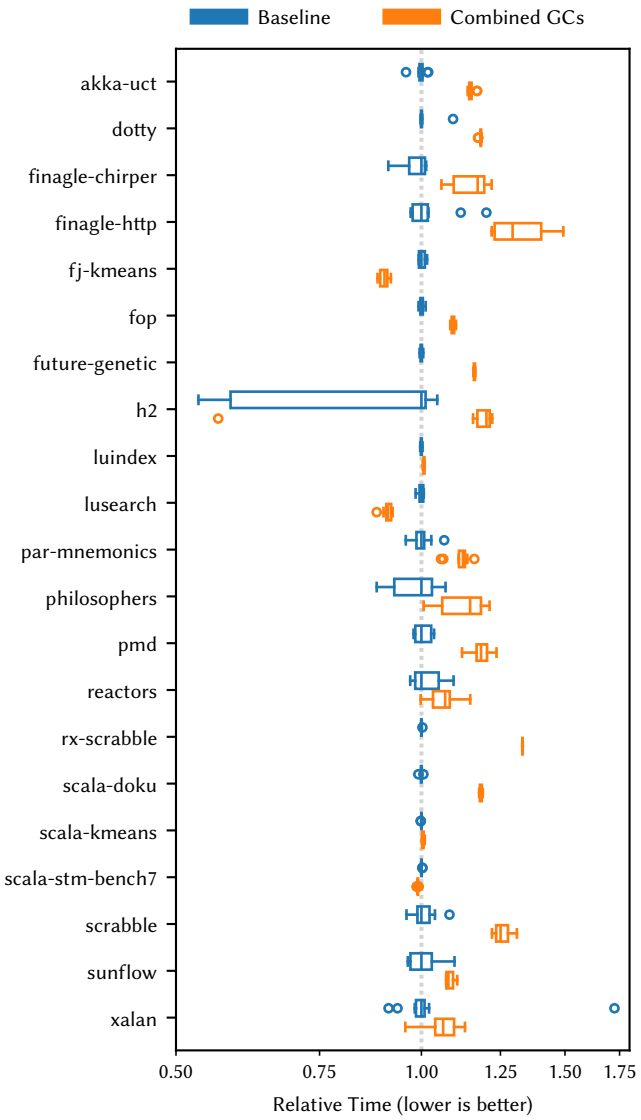

Figure 6. Execution times with Serial GC in the combined executable. As baseline, we use an executable containing only the Serial GC. The performance regression is expected and on average it is 11 % (min. −10 %, max. 33 %).

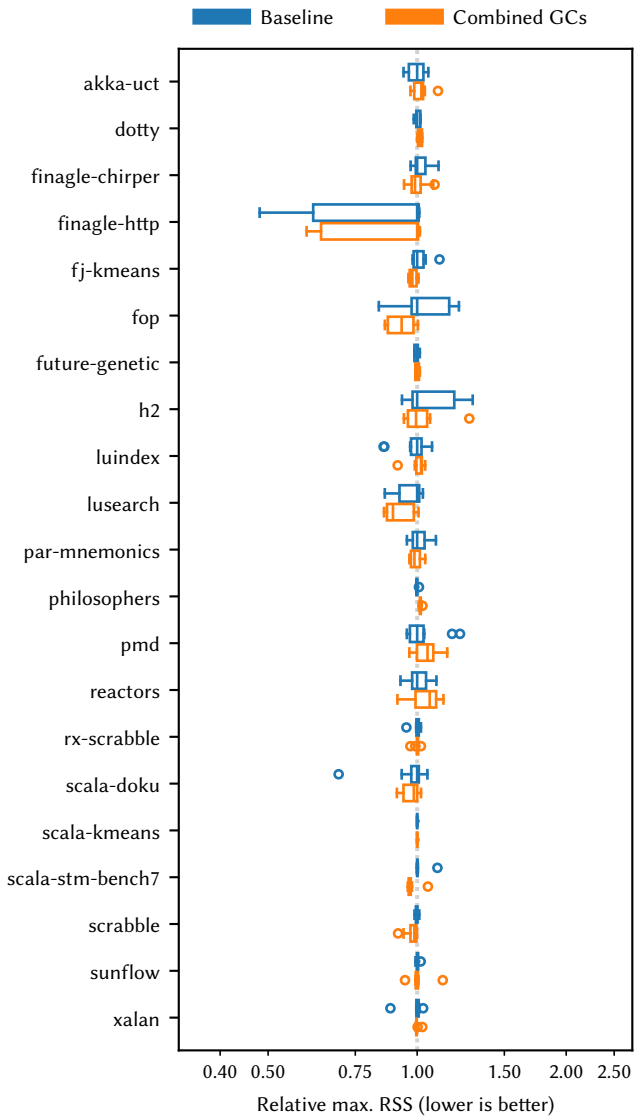

Figure 7. Max. RSS with G1 in the combined executable. The baseline is an executable containing only G1. On average, we see no performance change (min. −11 %, max. 6 %).

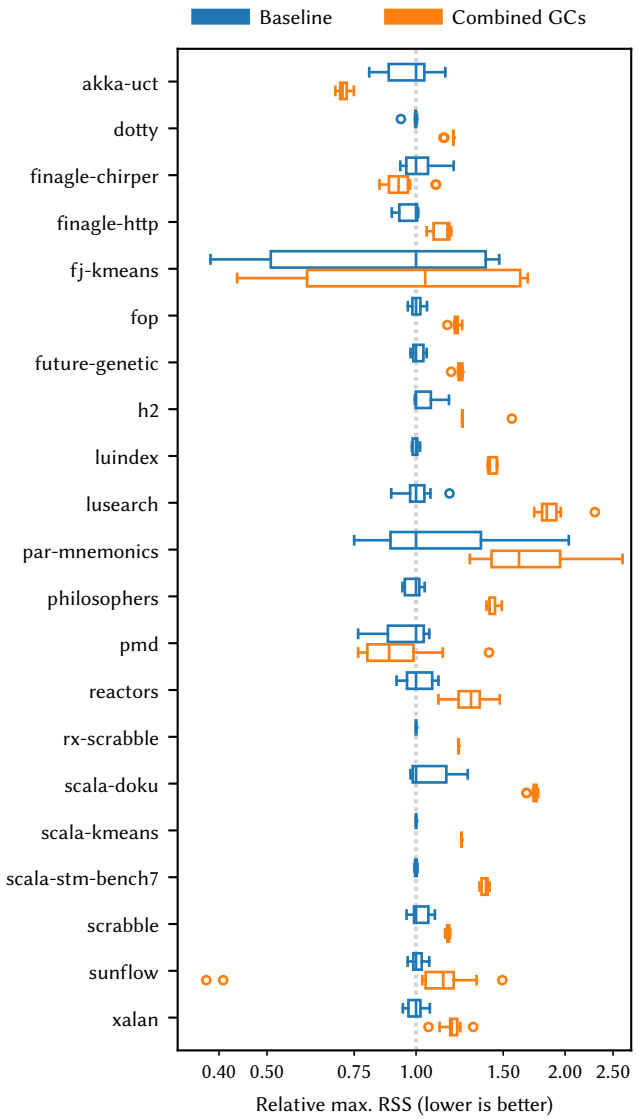

Figure 8. Max. RSS with Serial GC in the combined executable. The baseline is an executable containing only the Serial GC. The max. RSS regression is expected and on average it is 22 % (min. −29 %, max. 84 %).

4.6 Discussion

We believe these performance results to be acceptable for our system. The main goal was enable users to select the GC at run time on a per isolate basis by having both GC in the executable using the same AOT-compiled code. G1 is restricted to run in a single isolate, as it is directly derived from the HotSpot implementation. Now, we can use G1 in scenarios that require multiple isolates, which was not possible before. For the use case of Truffle languages illustrated in Figure 1, the benefit of being able to use G1 also outweighs the reduced performance of Serial GC.

5 Generalizability

One important remaining question for us is whether this approach is applicable more broadly. Unifying parts of GCs is as we saw possible for some GCs, but is not necessarily possible for all GCs one could conceive off. In this section, we briefly describe the GCs included in OpenJDK [20] and how applicable we think our approach is for them. Specifically, we discuss Epsilon, Serial, Parallel, G1, Shenandoah, and ZGC. We will focus on barriers for this analysis, since they are most critical for performance and functionality.

Brief Overview. The *Epsilon GC* does not collect anything and thus, does not need any barriers [20].

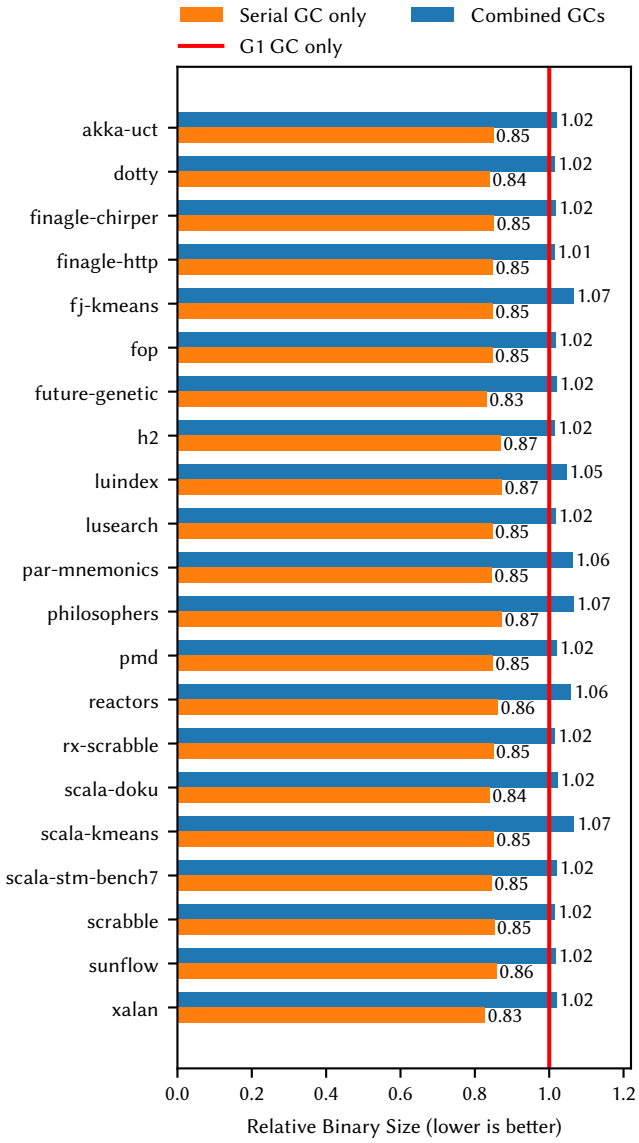

Figure 9. Relative binary size for executables containing the Serial GC or containing both GCs compared to executables containing only G1. On average, executables with both GCs are 3 % larger.

The *Serial GC* is a non-parallel, non-concurrent, generational GC. The *Parallel GC* is similar to the Serial GC, but uses multiple threads for collecting. Both GCs use the same write barriers for their card-table-based remembered set [20].

G1 also uses a card-table-based remembered set and corresponding write barriers. Additionally, it uses SATB barriers for concurrent marking. G1 needs read barriers only in certain cases, e.g., for reading weak references. The used read barrier is the same as the G1 SATB barrier [3, 20].

Shenandoah is a parallel, non-generational,[3] concurrent, compacting GC. It uses a conditional read barrier. If the GC is active and the object is in the current collection set, the barrier resolves a forwarding pointer or evacuates the object and sets the forwarding pointer. While it uses SATB barriers, it does not use a remembered set [6, 11, 12, 20].

ZGC is a mostly-concurrent, generational, parallel, compacting GC. It uses colored pointers that store additional information in the pointer. ZGC uses read barriers to ensure that the object is in a safe and usable state. It removes the color from the pointer and potentially updates pointers to relocated objects. Write barriers color the pointers and maintain the precise bitmap-based remembered sets. Additionally, ZGC uses SATB marking [10, 20, 24, 33].

Considerations for Unification. Table 1 contains a summary of the different barriers used by the different GCs. An *R* indicates that a GC requires this barrier. The implementations for the GCs may differ in the details. So, unifying multiple required barriers may have a performance impact. The ~ indicates that a GC normally does not need this barrier, but can use a unified version when necessary, again possibly having a performance impact. The × indicates that the barrier cannot be unified with our approach such that the GC supports it. Multiple GCs can be included in the same executable using our unifying approach, as long as they do not require a barrier that another GC cannot use sensibly, i.e., they cannot have an R and a × in the same column. For example, the Serial GC and the Parallel GC can be combined with our approach as they both require only a write barrier for a card-table-based remembered set. Whereas, the Serial GC and the ZGC probably cannot be combined with our approach as the Serial GC requires a card-table-based write barrier which cannot be unified sensibly for the ZGC.

As Epsilon does not use any barriers, it is a relatively simple case for unification. The barriers could be similar to our SATB barrier, which sees concurrent marking as being disabled and thus is skipped. Otherwise, the barrier might access a non-existent data structure if the Epsilon GC is used at run time. For example, the G1 write barrier dirties the card table if a reference write is region crossing and the written reference is not null. So, without an additional check, the Epsilon GC would have to use a dummy card table. This also holds for Shenandoah as it also does not use a card-table-based remembered set.

For including the Serial and the Parallel GC in the same executable, our approach is suitable as they already use the same barriers. In this paper, we have demonstrated that our approach is applicable for including G1 and the Serial GC in the same executable. Therefore, including G1 and the Parallel GC or all three GCs is also possible.

[3]There exists a generational Shenandoah variant, but in our analysis we only considered the default variant, which is the non-generational one.

Table 1. Barrier summary for OpenJDK's GCs. It shows which barrier a GC requires (R), can use a unified variant (~), and where unifying is not sensible (×). For example, the Serial GC could be combinable with a ZGC read barrier as indicated by the ~, however it requires a card-table-based write barrier as indicated by the R in the column, but this barrier cannot be sensibly unified with ZGC, as indicated by the ×. So, combining the Serial GC and ZGC is not plausible.

	Read Barrier		Write Barrier		
	Shenandoah	ZGC	SATB	Card	ZGC
Serial GC	~	~	~	R	×
Parallel GC	~	~	~	R	×
G1 GC	~	~	R	R	×
Epsilon GC	~	~	~	~	~
Shenandoah	R	×	R	~	~
ZGC	×	R	R	×	R

Including Shenandoah and any of the other GCs, excluding ZGC, in the same executable should be possible with our approach. Whether this combination is useful depends on the performance impact caused by the read barriers. As read barriers are more common than write barriers, they tend to have a larger performance impact.

Using our approach for including ZGC and any other GC, except Epsilon GC seems less plausible. The card-table-based write barriers used by Serial, Parallel and G1 GC seem to be too different to unify sensibly with the ZGC write barrier. Additionally, the load barriers of Shenandoah and ZGC are quite different, which make it also unclear that they would unify well. Creating uniform barriers would either require big changes to the GCs or would be close to a naive implementation of including both barriers.

Our approach requires not only uniform barriers, but also a uniform object layout, which includes the reference size. All investigated GCs except ZGC support both compressed and uncompressed references. ZGC only supports uncompressed references as it uses some bits in the pointers for coloring. So, if one would include ZGC and another GC in the same AOT-compiled executable, both would be limited to use uncompressed references with our approach.

6 Related Work

This section compares our approach to other work that supports run-time switching of GCs in a JIT compiling system, investigates different GC-agnostic barrier implementations in an AOT-compiling system, or that uses shared work packets for implementing GCs.

Just-In-Time-Compiled Systems. Soman et al. [28] extended JikesRVM to dynamically switch between five GCs at run time with a switching strategy guided by annotations in application code. Since they aim to support not just the GC selection, as we do, but switching between GCs, they need to, for instance, reserve separate bits in the object header for each GC. This is to support the switching where both GCs need to be able to access their information during the handover. We can however use the same bits for all GCs.

Another difference is that JikesRVM JIT compiles application code and inlines for instance GC-specific code into hot methods. When switching to another GC, they invalidate all methods with GC-specific code to trigger a recompilation, using on-stack replacement when necessary.

Furthermore, their system has two generational GCs that need write barriers. For methods compiled without optimization, they always insert this write barrier, which one could see as a uniform approach. Though, for methods compiled with optimization, they use their invalidation approach and insert the write barrier only for the generational GCs. Since we do not support run-time compilation, and a GC is chosen for the whole run time of an isolate, we instead needed to carefully unify the write barrier for both our GCs.

Ahead-of-Time-Compiled Systems. Hübner [8], Emdad [4], and Seoane Ampudia [27] worked on GC-agnostic barriers for AOT compilation with project Leyden. They support Serial GC, Parallel GC, G1, and ZGC.

Hübner [8] focused on read barriers. They proposed a patching barrier using self-modifying code. The emitted machine code of the fast path contains dummy operands or instructions, that get patched with the correct value for the used GC before execution. Our approach uses unified barriers. So, we can use multiple isolates with different GCs within the same OS process, while mapping the AOT-compiled code only once, which is important to limit overall memory use.

Emdad [4] created a unified fast path for their write barrier and kept the GC-specific slow path calls. Within the fast path, they use thread-local variables and branches to determine what operations the barriers have to perform, e.g., card marking. This affects the performance of all included GCs. We avoided it on the fast path and only have a check on the slow path. Additionally, they have removed the filtering from the card marking, which we kept in our unified version to not affect G1's performance.

Seoane Ampudia [27] proposed an asynchronous write barrier. The fast path records GC-agnostic information about all reference writes in a buffer. Once this buffer is full, the runtime processes all writes in a GC-specific manner.

Overall, all three approaches have also the goal of allowing users to select a GC at run time for AOT-compiled code, however, they only focus on the barriers. Instead, our approach also takes the object layout and the object header into account. Additionally, our implementation allows building native executables and selecting the used GC on a per isolate basis, allowing the use of different GCs at the same time within the same OS process.

Work Packets. Zhao et al. [36] proposed a new approach for implementing GCs using work packets. A work packet

encapsulates work items, a kernel, and scheduling dependencies. The work items are the smallest unit of work of a given type, e.g., marking the reference to a heap object. The kernel is the function that processes the work items. The scheduling dependencies specify what other work packets need to be processed before this work packet. These work packets can be used by different GC implementations.

They implemented this approach in the Rust MMTk GC framework and created 23 common packet types used by eight collectors. Some GCs use additional work packets that are unique to that GC. Additionally, they implemented work stealing of work packets and work items for better load balancing between multiple GC threads.

Although they did not investigate including multiple GCs in the same AOT-compiled binary, sharing the implementation using work packets may be useful for our approach. The GC parts that have to be unified due to the same AOT-compiled code can use the same work packets while the other parts can keep their unique work packets where necessary. The AOT-compiled executable could contain all work packets used by the included GCs. At run time only the ones used by the current GC get executed.

7 Limitations and Future Work

Our implementation is specific to the Serial GC and G1. For including another GC, the unified implementation of the GC classes, e.g., `UniversalHeap`, need to be extended. Additionally, using different unified barriers or object layout might be beneficial. For example, when only the Epsilon and the Serial GC are in the same executable, using the Serial GC write barriers avoids the performance penalty caused by the G1 write barriers.

Currently, the developer has to select the used GC with a run time option. In the future our implementation could be expanded to automatically select the GC based on the available hardware or profiling data.

8 Conclusion

In this paper, we presented a simple approach using unified barriers, object layout, object header and dynamic dispatching for including multiple GCs in the same AOT-compiled native executable and selecting the GC at run time. We implemented this approach in GraalVM Native Image for including the G1 and the Serial GC in the same executable. We changed the Serial GC to work with the G1-specific parts and left G1 itself unchanged. Using unified components allows multiple isolates to use different GCs, while mapping the AOT-compiled code only once.

The geometric mean over all benchmarks shows no change in peak performance and max. RSS for G1, when comparing our implementation, with G1 and the Serial GC in the same executable, to executables containing only G1. For the Serial GC in the combined executable the geometric mean shows a peak performance regression of 11 % (min. −10 %, max. 33 %) and a 22 % max. RSS regression (min. −29 %, max. 84 %) when comparing to executables containing only the Serial GC. The regressions are caused by changing the Serial GC to use the G1-specific parts, e.g., the larger G1 write barrier. Including two GC in the same executable increases the binary size by 3 % compared to executables containing only G1.

Acknowledgments

This research project was partially funded by Oracle.

References

[1] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, 940–954. doi:10.1145/3669940.3707217

[2] C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798

[3] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, 37–48. doi:10.1145/1029873.1029879

[4] Arveen Emdad. 2025. *Garbage Collector-Agnostic Barriers for Ahead-of-Time Compilation.* Master's thesis. Uppsala University, Department of Information Technology.

[5] Robert R. Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (Nov. 1969), 611–612. doi:10.1145/363269.363280

[6] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, Article 13, 9 pages. doi:10.1145/2972206.2972210

[7] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. 1992. A comparative performance evaluation of write barrier implementation. *SIGPLAN Not.* 27, 10 (Oct. 1992), 92–109. doi:10.1145/141937.141946

[8] Paul Hübner. 2025. *Garbage Collector Agnostic Load Barriers : Moving Towards Flexible Ahead-of-Time Java Compilation.* Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[9] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (2nd ed.). Chapman and Hall/CRC. doi:10.1201/9781003276142

[10] Stefan Karlsson. 2024. *JEP 439: Generational ZGC.* https://openjdk.org/jeps/439

[11] Roman Kennke. 2019. *Shenandoah GC in JDK 13, Part 1: Load reference barriers.* Retrieved February 24, 2026 from https://developers.redhat.com/blog/2019/06/27/shenandoah-gc-in-jdk-13-part-1-load-reference-barriers

[12] Roman Kennke. 2019. *Shenandoah GC in JDK 13, Part 2: Eliminating the forward pointer word.* Retrieved February 24, 2026 from https://developers.redhat.com/blog/2019/06/28/shenandoah-gc-in-jdk-13-part-2-eliminating-the-forward-pointer-word

[13] Oracle Corporation. 2025. *Native Image.* Retrieved November 25, 2025 from https://www.graalvm.org/jdk25/reference-manual/native-image/

[14] Oracle Corporation. 2025. *Native Image Basics.* Retrieved November 25, 2025 from https://www.graalvm.org/jdk25/reference-manual/native-

image/optimizations-and-performance/MemoryManagement/

[15] Oracle Corporation. 2025. *Native Image Basics.* Retrieved November 25, 2025 from https://www.graalvm.org/jdk25/reference-manual/native-image/basics/

[16] Oracle Corporation. 2026. *A high-performance embeddable JavaScript runtime for Java.* Retrieved March 2, 2026 from https://www.graalvm.org/javascript/

[17] Oracle Corporation. 2026. *A high-performance embeddable Python 3 runtime for Java.* Retrieved March 2, 2026 from https://www.graalvm.org/python/

[18] Oracle Corporation. 2026. *A high-performance embeddable WebAssembly runtime for the JVM.* Retrieved March 2, 2026 from https://www.graalvm.org/webassembly/

[19] Oracle Corporation. 2026. *GraalVM.* Retrieved January 15, 2026 from https://github.com/oracle/graal

[20] Oracle Corporation. 2026. *jdk.* Retrieved February 20, 2026 from https://github.com/openjdk/jdk/

[21] Oracle Corporation. 2026. *Profile-Guided Optimization.* Retrieved December 04, 2025 from https://www.graalvm.org/latest/reference-manual/native-image/

[22] Oracle Corporation. 2026. *Truffle Language Implementation Framework.* Retrieved March 2, 2026 from https://www.graalvm.org/jdk25/graalvm-as-a-platform/language-implementation-framework/

[23] Oracle Corporation. 2026. *Truffle Language Implementation Framework.* Retrieved March 2, 2026 from https://github.com/oracle/graal/tree/master/truffle

[24] Erik Österlund. 2026. *The Z Garbage Collector: In JDK 25* (1st ed.). Chapman and Hall/CRC. doi:10.1201/9781003595366

[25] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, 31–47. doi:10.1145/3314221.3314637

[26] Thomas Schrott. 2025. *Supporting Multiple Garbage Collectors in a Single GraalVM Native Executable.* Master's thesis. Johannes Kepler University Linz.

[27] Antón Seoane Ampudia. 2025. *Garbage Collector-Agnostic Store Barriers for Ahead-of-Time Compilation: An Out-of-Line Path Towards Java Future.* Master's thesis. Uppsala University, Department of Information Technology.

[28] Sunil Soman, Chandra Krintz, and David F. Bacon. 2004. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, 49–60. doi:10.1145/1029873.1029880

[29] Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira. 2023. BestGC: An Automatic GC Selector. *IEEE Access* 11 (2023), 72357–72373. doi:10.1109/ACCESS.2023.3294398

[30] Christian Wimmer. 2019. *Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM.* Retrieved January 15, 2026 from https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e

[31] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (Oct. 2019), 29 pages. doi:10.1145/3360610

[32] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, 187–204. doi:10.1145/2509578.2509581

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

[34] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *Proceedings of the 2012 International Symposium on Memory Management* (Beijing, China) *(ISMM '12)*. Association for Computing Machinery, 37–48. doi:10.1145/2258996.2259004

[35] T. Yuasa. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.* 11, 3 (March 1990), 181–198. doi:10.1016/0164-1212(90)90084-Y

[36] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2025. Work Packets: A New Abstraction for GC Software Engineering, Optimization, and Innovation. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 361 (Oct. 2025), 29 pages. doi:10.1145/3763139