Improving Native-Image Startup Performance

Matteo Basso
Università della Svizzera italiana (USI)
Lugano, Switzerland
matteo.basso@usi.ch

Aleksandar Prokopec
Oracle Labs
Zurich, Switzerland
aleksandar.prokopec@oracle.com

Andrea Rosà
Università della Svizzera italiana (USI)
Lugano, Switzerland
andrea.rosa@usi.ch

Walter Binder
Università della Svizzera italiana (USI)
Lugano, Switzerland
walter.binder@usi.ch

Abstract

With the increasing popularity of Serverless computing and Function as a Service—where typical workloads have a short lifetime—the research community is increasingly focusing on startup performance optimization. To reduce the startup time of managed language runtime systems, related work proposes strategies to move runtime environment initialization ahead-of-time. For instance, GraalVM Native Image allows one to create a binary file from a Java application that embeds a snapshot of the pre-initialized heap memory and can run without instantiating a Java Virtual Machine. However, the program startup needs to be further optimized, because the cloud runtime often starts the program while responding to the request. Thus, the program startup time impacts the service-level agreement.

In this paper, we improve the startup time of Native-Image binaries by changing their layout during compilation, reducing I/O traffic. We propose a profile-guided binary-reordering approach and a profiling methodology to obtain the execution-order profiles of methods and objects. Thanks to these profiles, we first reduce page faults related to the code section. Then, we propose three ordering strategies to reduce page faults related to accessing the objects in the heap snapshot. Since the object identities and the heap-snapshot contents are not persistent across Native-Image builds of the same program, we propose a method of matching objects from the profile against the objects in the profile-guided build. Experimental results show that our ordering strategies lead to an average page-fault reduction factor of 1.61× and an average execution-time speedup of 1.59×.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software performance**; *File systems management*; *Virtual machines*; • **Computer systems organization** → **Cloud computing**.

Keywords: GraalVM, Native Image, Startup Performance, Profiling, Serverless Computing, Function as a Service.

ACM Reference Format:
Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2025. Improving Native-Image Startup Performance. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3696443.3708927

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708927

1 Introduction

In contrast to long-running server-side workloads where steady-state performance is crucial, modern short-running workloads—typically executed on Serverless and Function as a service (FaaS) cloud-computing services—incur significant overheads when the runtime relies solely on Just-In-Time (JIT) compilation. Indeed, JIT compilation enables high steady-state performance but introduces runtime and memory overheads, which affect program startup [2]. For this reason, recent research is increasingly focusing on the optimization of startup performance, cloud lambda functions [55], and interpreters [6]. Improving startup performance of short-running applications is crucial to save computational resources, maximizing the throughput of cloud services.

In the Serverless and FaaS computing models [17], the service needs to balance between keeping programs in memory and starting programs too often. When a certain machine receives a request for the first time, the service needs to prepare the execution environment with the required memory, runtime, and configuration to run the user-provided function on that machine. The code of the function can be either fully downloaded in this setup step or incrementally downloaded using a Network File System (NFS), upon the first function execution. Since the environment is already initialized and the function code is already present in the RAM, subsequent function invocations are significantly faster than the first one. However, to avoid wasting resources, the service typically retains the execution environment only for a certain period of time [4]. After that, the service frees the resources

by removing the idle program, and a new function request may later start the initialization from scratch, incurring in additional overhead. The service would like to remove the idle programs from main memory as soon as possible, but without breaking the service-level agreement that a certain percentile of responses takes less than a certain number of milliseconds. Improving the program startup time allows the service to remove idle programs more often.

While related work on optimizing start-up performance focuses mostly on the optimization of the Serverless platforms [55], a few techniques try to perform startup optimizations at the application level. For instance, GraalVM Native Image [57] allows creating a binary file from a Java application that can run without instantiating a Java Virtual Machine (JVM), pre-initializing at build time the Java environment. Binaries produced by Native Image contain not only the code to be executed, but also a snapshot of the pre-initialized heap memory, consisting of Java objects and arrays. While embedding this snapshot reduces the runtime-initialization time, the larger binary size increases the pressure on the (Network) File System. Hence, even when employing these techniques, startup performance is not optimal.

This paper aims at mitigating startup-performance degradation for the first execution of binaries that embed a snapshot of the heap memory. We propose a profile-guided methodology to reorder the code and the heap-snapshot sections of the binary (Sec. 3). We generate an instrumented binary of the program to collect a *profile*, containing a trace of method invocations (which reflects the order in which they were first executed) and a trace of accesses to objects in the heap snapshot (which reflects the order in which they were accessed). Using these profiles, we create a second, profile-driven optimized binary. We use the traces to place the used methods and objects into contiguous areas of the binary. While the invocation traces can be mapped to the methods in the optimized binary by matching their signatures, mapping the object-access traces to the heap snapshot is more challenging. Since a heap object does not have a unique name or identifier, and the heap-snapshot contents are not guaranteed to be the same across image builds (due to non-determinism in running class initializers and because profiles themselves influence the contents of the binary), the object-access trace needs to be mapped to respective objects using other distinguishing factors. We are not aware of other work dealing with the ordering of heap snapshots stored in binary files, mapping object identities across builds that differ due to divergence between the regular and the profile-driven image.

In addition, we propose multiple ordering strategies aiming at reducing page faults related to accesses to the binary. We first describe two code-ordering strategies, which improve runtime performance and locality in the code section of the binary (Sec. 4). One strategy is based on ordering compilation units, while the other relies on method ordering. Then, we propose three heap-ordering strategies to reduce page faults related to accessing objects in the heap snapshot, as well as a novel way of mapping profiles against objects in the heap (Sec. 5). One strategy relies on incremental identifiers, another on structural hashing and the third encodes paths in the heap object graph. We also propose a profiling methodology to collect the profile (Sec. 6).

Finally, we evaluate our code- and heap-ordering strategies on the "Are We Fast Yet?" (AWFY) benchmark suite [33] and three widely-used microservice frameworks, i.e., *micronaut* [34], *quarkus* [47], and *spring* [53] (Sec. 7). Experimental results show that our ordering strategies are effective in both reducing page faults and improving runtime performance, leading to an average page-fault reduction factor of 1.65× and 1.46× and an average execution-time speedup of 1.59× and 1.61× on AWFY and microservices, respectively.

We complement the paper by illustrating the required background (Sec. 2), discussing related work (Sec. 8), and giving our concluding remarks (Sec. 9).

2 Background

In the following text, we give preliminary information on ahead-of-time compilation, heap snapshotting, and profile-guided optimizations.

Ahead-of-time (AOT) Compilation To reduce the startup time of Java workloads, GraalVM Native Image [39, 57] (henceforth just *Native Image* for short) allows compiling a JVM application and its dependencies into a single binary file that can be executed without instantiating a JVM. To do so, Native Image relies on Graal [13], an optimizing compiler that performs inlining [46], escape-analysis [51], and various other optimizations [26, 27]. Graal performs transformations and optimizations on a portion of code provided as input, called *compilation unit* (CU). A CU consists of a *root method* (i.e., the method from which the compilation started), and all the methods that were inlined into that root method. CUs are stored in the `.text` section of the binary. After the compilation, each CU typically includes multiple inlined methods. By default, CUs in the `.text` section of a Native-Image binary are ordered alphabetically. Notably, the CUs in one binary may not correspond to the CUs in another binary of the same compiled application. The contents of a Native-Image binary are sensitive to the code that is on its classpath. Indeed, Native Image uses a form of *points-to analysis* to decide which code from the classpath is reachable [22, 22, 49, 57, 58], and to improve compilation speed, it employs *saturation* to mark virtual calls as having all possible targets after the set of targets exceeds a specific threshold [58]. The points-to analysis is conservative and always includes more code than what is actually reachable or executed at runtime. The inclusion of seemingly unrelated code into the binary may thus significantly impact inlining decisions, hence producing a completely different grouping of Java methods into compilation units. Inlining decisions are furthermore code-size

driven, so instrumentation code may make the inliner behave differently between compilations of the instrumented and the regular image.

Heap Snapshotting A defining feature of Native Image is that, to further speed up the startup, the produced binaries contain a snapshot of the Java heap memory. The snapshot is obtained after executing the static initializers of the classes that are deemed to be reachable in the startup process of the VM (when static initializers have no observable side-effects). The aforementioned points-to analysis determines which classes and static fields are reachable. To select the objects to be included in the heap snapshot, Native Image traverses the object graph in a well-defined order, starting from the required static fields of the reachable classes, as well as constants in the code section. For this reason, small changes in the program or its entry points may lead to significant changes in the heap snapshot. Moreover, objects in the heap snapshot typically differ across Native-Image compilations, particularly when the second compilation consumes profiles to guide its optimizations. For example, due to different inlining decisions that affect Partial Escape Analysis (PEA) [51], some objects could be stack-allocated in one binary but not in another, or the accesses to their fields could be constant-folded, eliminating the need to store the respective objects in the heap snapshot. The heap snapshot is stored in the `.svm_heap` section of the binary, and is memory-mapped when the program starts, hence each page is lazily copied to memory on the first access. By default, objects are ordered according to the order of the CUs in the `.text` section of the binary—objects reachable from a CU A are stored before objects reachable from another CU B that is stored after A in the `.text` section. We note that the compilation is in some cases non-deterministic, and one reason is that the class initializers may be executed in parallel during the build process.

Profile-guided Optimizations (PGO) Native Image can use execution profiles to generate more efficient code, and this is yet another reason for inconsistencies between regular and profile-driven builds. As is common for AOT compilers such as LLVM [31] and GCC [16], Native Image can create an *instrumented binary* with code that gathers profiles, and writes them to a file upon program exit. Native Image can then use the profiles to generate an *optimized image*. Native-Image profiles currently contain branch frequencies, virtual-call receiver types, and method call counts. Instrumented and optimized images differ in their CUs and objects in the heap snapshot, which is primarily caused by different inlining decisions that enable different sets of optimizations.

3 Profile-Guided Binary Reordering

Our goal is to improve the existing profiles collected by instrumented Native-Image binaries, and use the augmented profiles to generate an optimized binary with improved startup performance. Fig. 1 shows the steps required by our methodology and how they are integrated into Native Image. The figure reports both the steps required to create the instrumented binary in the profiling build (gray nodes with dotted borders) and the steps required to create the optimized image introduced by our methodology (blue nodes with dashed borders). Steps required for both the profiling and the optimized builds are depicted with green nodes with dash-dot borders. White nodes with solid borders represent existing steps of the Native-Image building process, while white nodes with double borders represent outputs.

The regular Native-Image building process starts with the iterative execution of a *points-to analysis* [22, 22, 49, 57, 58] to run static initializers and create a snapshot of the heap until a fixed point is reached. Then, Native Image compiles the reachable methods, adding their code to the `.text` section of the binary, and stores the heap snapshot in the `.svm_heap` section of the binary. To produce instrumented binaries in the profiling build, our methodology extends the regular building process to 1) instrument the compiled methods to collect method-execution and object-access traces, and 2) associate an identifier to each object instance to be stored in the `.svm_heap` section of the binary (detailed later in Sec. 5). The execution of the instrumented binary leads to the generation of traces that need to be further post-processed to produce the actual code- and heap-ordering profiles.

In the optimizing build, our methodology exploits the ordering profiles gathered upon the execution of the instrumented binary. We add a code-ordering step that takes the code-ordering profiles as input (note that the dashed arrow in the figure connects the code-ordering profiles to the code-ordering step) and reorders the CUs before storing them into the binary. Moreover, we add the same step present in the profiling build to associate an identifier to each object instance in the heap snapshot, and a heap ordering step that takes the heap-ordering profiles as input, before writing the heap snapshot. The heap-ordering step attempts to match the semantically same objects in the heap snapshot and in the profiles by exploiting their identifiers and hence reorders the former according to the latter. In this build, identifiers are not stored in the binary.

4 Code Ordering

In our approach, we extend the instrumentation of the program to collect the trace of method invocations, which records the order in which the methods executed. The CUs in the `.text` section of the optimized binary are then reordered using the trace, with the goal of ordering and minimizing the total set of pages with the code from the trace, and thus to reduce the number of page faults.

We order CUs with the aim of achieving the following optimality property as often as possible:

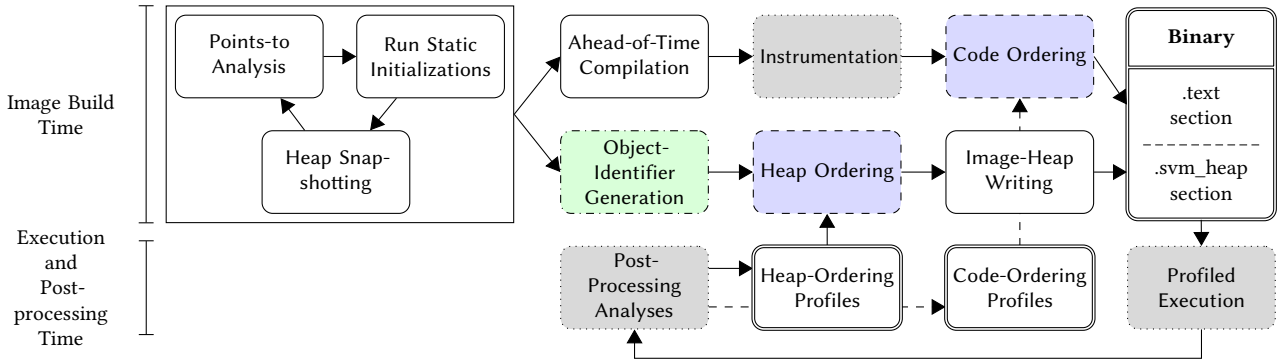


Figure 1. Integration of the proposed binary-reordering methodology into the Native-Image building process.

Property 1 (Optimal ordering)**.** If the first invocation of method *A* appears in the trace before the first invocation of method *B*, then the first occurrence of the method *A* precedes the first occurrence of the method *B* in the layout of the optimized binary.

In general, for any method-invocation trace of a program and any choice of CUs, it is not possible to choose a CU ordering that satisfies Property 1. The reason for this is that the same method from the trace may have been invoked on multiple code paths. Consider the following example:

```
a() {                  b() {                  c() {
  if (...) b();          if (...) c();          ...
  if (...) c(); }      }                      }
```

Next, consider the following choice of compilation units: the call from `a` to `b` is inlined into `a`, the code from `b` to `c` is also inlined into `a`, but the call from `a` to `c` is not inlined (i.e., method `c` remains a separate CU). Finally, consider the method-invocation trace `a, b, c`. It is not possible to decide whether the inlined or the non-inlined invocation of `c` occurred in the trace. Hence, when laying out the CUs in the binary, it is not clear whether it is optimal to place the CU with the root `c` immediately after the CU with the root `a`.

We therefore implement and evaluate two code-ordering heuristics: one that orders the CUs based on the invocation orders of the root methods, and another that orders the CUs based on the invocation orders of all the methods.

4.1 Compilation-Unit-Based Ordering

In this strategy (called *cu ordering*), we trace all the CU executions by instrumenting the CU entry points. In particular, the instrumentation code records the signature of the root method of each CU. To obtain the ordering profiles, we remove the duplicated elements in the trace maintaining the original (execution) order.

Consider the previous example and the execution of the following code `a()` in the case where the conditions of all the `if` statements evaluate to `true`. This strategy produces the CU-invocation trace `a, c` (since `c` is not inlined into `a`), leading to the ordering `a, c`.

4.2 Method-Based Ordering

In this strategy (called *method ordering*), we trace all the method executions by instrumenting the method entry points. The instrumentation code records the signature of each method. To obtain the ordering profiles, we remove the duplicated elements in the trace maintaining the original (execution) order. In the same example presented in Sec. 4.1, this strategy would produce the method-invocation trace `a, b, c, c`, leading to the ordering `a, b, c`. This ordering is convenient if the inliner decide to not inline method `b` in method `a` in the optimized binary.

5 Heap-Snapshot Ordering

In this section, we propose and describe three heap-ordering strategies, aiming at reducing page faults related to the `.svm_heap` section of the binary. The goal of each strategy is to compute 64-bit object identities (IDs) to match the object-access trace entries with the heap-snapshot objects of the optimized binary as accurately as possible. The first strategy does this by assigning sequential IDs to objects in the order in which they are encountered during heap snapshotting (Sec. 5.1); the second strategy identifies objects through hashes, computing them taking into account the type and the fields of the object, as well as its neighbours in the object graph (Sec. 5.2); the third strategy assigns IDs to objects based on the path to the heap object (Sec. 5.3).

5.1 Incremental ID

Here, we propose a strategy that assigns incremental IDs to object instances in object encounter order when traversing the heap object graph to detect objects to be included in the heap snapshot. This strategy called *incremental ID*, has the advantage of being simple, but it becomes inaccurate for complex workloads whose code and heap snapshots differ between regular, profiling, and optimized builds.

Algorithm 1 shows the pseudocode of the proposed strategy. All algorithms shown in the paper take as input an *entity*, which represents a wrapper around a *value*, which

Algorithm 1: Incremental IDs Function

```
Function incrementalId(entity):
computes the 64-bit ID for the value wrapped by entity
using incremental IDs
Input:
entity, a wrapper around the value for which the algorithm
computes the ID
Output:
the 64-bit ID for the value wrapped by entity
1  if entity.isNull() then
2      return 0
3  type ← entity.type()
4  typeId ← type.id()
5  id ← getCounterFor(typeId).incrementAndGet()
6  return (typeId << 32) | id
```

can be an object reference, an array reference, or a primitive value. The purpose of *entity* is storing and inspecting metadata of the wrapped *value*. We anticipate that primitive value wrappers will only be encountered in function `encodeToBytes` (Algorithm 2, explained later), because IDs need to be computed only for objects and arrays. We generate 64-bit IDs, where the most-significant 32 bits store a unique ID associated with the type (lines 3–4, 6) while the least-significant 32 bits store an incremental ID associated to the ID of the type of the value wrapped by *entity* (lines 5–6). That is, objects have incremental IDs within their type, not globally. Doing so helps reduce inaccuracies due to different object encounter orders among different compilations because in this way the inaccuracies introduced by an object affect only the ordering of the objects of the same type and not the ordering of all the objects.

We note that types can be uniquely identified by their fully qualified names even between compilations and hence are easily associated with IDs. The helper function `getCounterFor` (line 5) returns a counter object instance associated with the provided type ID. If a type ID has no counter associated, the function creates a new counter with an initial value of zero and associates it with the type ID.

5.2 Structural Hash

In this section, we propose a strategy that computes object IDs leveraging a structural hash function, i.e., a function that analyzes the object structure and hashes all its fields. We call this strategy *structural hash*. We note that we implement our own hashing function and we do not resort to the Java method `System.identityHashCode(Object)` (i.e., the default hash function implementation invoked by `Object.hashCode()`) because the hash computed on the semantically same object across compilations most likely differs, invalidating object mappings. Indeed, implementations of `System.identityHashCode(Object)`, which are platform-specific, often rely on either random values or the memory address at which the object was allocated. Similarly, we do not use as hash the one computed by the `hashCode` method because this method is not guaranteed to be declared for

Algorithm 2: Structural Hash Function

```
Function structuralHash(entity):
computes the structural hash for the value wrapped by
entity
Input:
entity, a wrapper around the value to be hashed
Output:
the 64-bit structural hash for the value wrapped by entity
1  bytes ← encodeToBytes(entity, 0)
2  return murmurHash3(bytes)

Function encodeToBytes(entity, depth):
encodes the value wrapped by the provided entity into a
byte buffer
Input:
entity, a wrapper around the value to be encoded
depth, the current recursion depth
Output:
a byte buffer that encodes the value wrapped by entity
1  bytes ← newByteBuffer()
2  if entity.isNull() then
3      bytes.append(0)
4      return bytes
5  bytes.append(entity.type().fullyQualifiedName())
6  shouldRecurse ← depth < MAX_DEPTH
7  if entity.isPrimitive() or entity.isString() then
8      bytes.append(entity.value())
9  else if entity.isObjectInstance() then
10     fields ← entity.fields()
11     for k ← 1 to fields.length() do
12         field ← entity.getFieldWrapper(k)
13         if shouldRecurse or field.isPrimitive() or
             field.isString() then
14             fieldTypeName ←
                 field.type().fullyQualifiedName()
15             bytes.append(fieldTypeName)
16             bytes.append(encodeToBytes(field, depth +
                 1))
17 else if entity.isArray() then
18     elementType ← entity.elementType()
19     bytes.append(elementType.fullyQualifiedName())
20     bytes.append(entity.length())
21     if shouldRecurse or elementType.isPrimitive() or
         elementType.isString() then
22         for k ← 1 to entity.length() do
23             bytes.append(k)
24             element ← entity.getElementWrapper(k)
25             bytes.append(encodeToBytes(element, depth+
                 1))
26 return bytes
```

all types or implemented efficiently, and can contain side-effects.

Our approach, shown in Algorithm 2, exploits metadata to hash object instances of arbitrary types. Function `structuralHash` first encodes the wrapped value in a byte buffer by exploiting the recursive `encodeToBytes` function (line 1, described later) and then leverages the widely used hash function *MurmurHash3* [1] (line 2), i.e., a fast hash function that produces well-distributed hash values, useful "in every scenario when we need to find two or more matching byte arrays" [1]. Encoding the wrapped value to bytes allows computing the hash on the entire data and avoids computing and merging partial hashes.

The recursive `encodeToBytes` function encodes an object with all its fields (it is recursively invoked when the field value is an object reference) and consists of four cases, explained below. In addition to the wrapper around the value, the function takes as input the current recursion depth (starting from 0) and produces a byte buffer as output. First, the algorithm initializes an empty byte buffer to store the bytes to be returned (line 1). If the wrapped value is `null` (line 2), the algorithm stores 0 in the buffer and returns it (lines 3–4). If the wrapped value is not `null`, the algorithm stores (in the buffer) the bytes representing the fully qualified name of the type of the value and checks whether the current depth exceeds a certain threshold `MAX_DEPTH` (lines 5–6). The resulting value of this check, stored in the `shouldRecurse` variable, will be later used to determine whether the algorithm should recurse or not when encountering a reference to an object instance or array. This is required to avoid infinite recursion since the object graph may contain cycles. The higher the value of `MAX_DEPTH`, the higher the computation time, the lower the collisions of the hash function but also the lower the probability of matching objects across compilations due to the inclusion of divergences between the heap snapshots in the hash.

Then, the algorithm computes the encoding based on the value type. If the value is of a primitive type or `String`, we simply append the primitive value or the bytes representing the string to the buffer, respectively (lines 7–8). If the value is an object instance, we iterate over the object fields (in source-code definition order) and we read the value stored in each field as an entity (lines 9–12). For each field, the algorithm checks whether it can recurse on the field entity or whether the dynamic type of the field value is a primitive type or `String` (line 13). If the check succeeds, we append the fully qualified name of the field's static type to the byte buffer (lines 14–15), as well as the bytes resulting from a recursive call that takes the field entity and the depth (incremented by 1) as parameters (line 16).

Finally, if the value is an array, we first append the fully qualified name of the array element type and the array length to the byte buffer (lines 17–20). Then, if the current depth allows recursion or the array element type is a primitive type or `String`, we iterate over all the wrapped array elements, appending for each of them the corresponding index within the array and the bytes resulting from the recursive encoding on the array element (lines 21–25). The algorithm terminates by returning the byte buffer (line 26).

Algorithm 3: Heap Path Hash Function

```
Function heapPathHash(entity):
computes the 64-bit hash for the value wrapped by entity
based on heap paths
Input:
entity, a wrapper around the value to be hashed
Output:
the 64-bit hash for the value wrapped by entity
1  if entity.isNull() then
2      return 0
3  bytes ← newByteBuffer()
4  if entity.isRoot() and
     entity.inclusionReason() == "InternedString" then
5      bytes.append(entity.value())
6  else
7      current ← entity
8      while true do
9          bytes.append(current.type().fullyQualifiedName())
10         if current.isRoot() then
11             bytes.append(current.inclusionReason())
12             break
13         else
14             parent ← current.getParents().first()
15             if current.isArray() then
16                 index ←
                     getAccessedArrayIndex(parent, current)
17                 bytes.append(index)
18             else
19                 field ←
                     getAccessedField(parent, current)
20                 bytes.append(field.descriptor())
21             current ← parent
22 return murmurHash3(bytes)
```

5.3 Heap Path

In this section, we propose a strategy named *heap path*. This approach uses as object ID a hash computed based on 1) the first path in the heap object graph to that object found by Native Image, i.e., the path that led to the inclusion of that object in the heap snapshot, and 2) the *heap-inclusion reason* associated with the root of that path. The heap-inclusion reason is a string representing why Native Image has deemed the root to be such. The heap-inclusion reason associated with a root object may be the signature of a static field (for an object stored in a static field of a reachable class), the signature of a method (for an object that is referenced by a

constant pointer embedded in a method), "InternedString" (for a Java interned string [40]), "DataSection" (for an object stored in the data section of the binary), or "Resource" (for an object representing a resource). The advantage of this strategy is that heap paths are less sensitive to divergences between compilations than incremental IDs (as shown later in Sec. 7.2). The disadvantage is that the same object may be reachable from multiple paths. Our strategy considers only the single path that led to the inclusion of that object in the heap snapshot at image build time, which may be different across compilations.

Algorithm 3 reports the pseudocode of the iterative *heap-path* hash function. The value wrapped in the input *entity* (for which the hash has to be computed), is the last object/array in the path that needs to be written in the `.svm_heap` section of the binary. Similarly to the *structural hash* strategy, this function returns a 64-bit hash computed via MurmurHash3.

The algorithm first checks whether the wrapped value is `null`, returning zero in such case (lines 1 and 2). If the value is not `null`, the algorithm allocates a byte buffer that will be later used by MurmurHash3 (line 3) and checks whether the value is a root in the heap object graph and whether this root was included in the heap object graph because it represents an interned string (line 4). If the value is an interned string, we do not hash the heap path (that would be the same for all the interned strings) but instead, we append the bytes representing the string to the buffer (line 5). If the value is not an interned string, we iteratively traverse the first path from the object to the root (lines 7–21). For each object in the path, we append the fully qualified name of the type of the object to the buffer (line 9).

If the value is a root, we append the heap inclusion reason as a string and we break the loop (lines 10–12). If the value is not a root, we obtain the parent object or array in the path (line 14), If the parent is an array (line 15), we obtain the array index where the current wrapped value is stored (line 16) and we append it to the buffer (line 17). Otherwise, the parent is an object instance. Hence, we obtain the field where the current wrapped value is stored (line 19) and we append the field descriptor to the buffer (line 20). We then iterate over the parent (line 21).

Finally, after processing all objects in the path, we apply MurmurHash3 and we return the hash (line 22).

6 Profiling Methodology

In this section, we detail our profiling methodology (Sec. 6.1) and post-processing analysis (Sec. 6.2).

6.1 Tracing Profiler

Our approach makes use of a tracing profiler, i.e., a profiler that produces a (per-thread) sequence of executed events. The profiler performs the instrumentation at the level of the intermediate representation (IR) [12, 13] used by the compiler during its optimization passes. We resort to this technique because, for our goals, it would be impractical to perform the instrumentation at other levels, such as machine code or bytecode. Instrumentation at machine-code level would lack additional metadata (obtained upon compilation) necessary to identify 1) all methods corresponding to machine-code instructions (needed to perform code ordering) and 2) machine-code instructions corresponding to Java object field and array accesses (needed to perform heap ordering). Instead, bytecode-level instrumentation would have severe drawbacks, as it would 1) disrupt the optimizations normally done by the compiler and 2) overprofile Java field and array accesses, leading to highly inaccurate ordering profiles [7].

In particular, our profiling methodology leverages an accurate IR-level path-profiling technique proposed by related work [7]. Using this technique, we accurately track executed events, lowering perturbation on compiler optimizations and increasing the accuracy of the profiles. Moreover, our tracing profiler exploits the path-cutting optimization to the standard path-profiling algorithm proposed by the same related work [7], which is fundamental to avoid an exponentially large number of paths and enables the practical usage of path profiling in a modern optimizing compiler. We implement the profiler within the Graal compiler.

Upon instrumentation, each application path is associated with a unique ID. We modify the technique proposed by related work, which performs event counting, to perform event tracing instead, i.e., we do not count path executions but we store the IDs associated to the executed paths into thread-local buffers, producing one trace file per thread. By iterating over the ordered path IDs in a trace file, one can obtain the entire sequence of events executed by a thread (such as the ordered object accesses performed by a thread). Instrumentation code is implemented as handcrafted IR nodes with some specific calls to low-level functions to obtain and dump thread-local buffers. These functions are implemented as methods with no heap accesses, allowing the collection of events occurring even during the initialization of the execution environment, when code cannot be interrupted and the heap memory has not been initialized yet. This is crucial to optimize not only the user code but also the Native-Image internals employed in the very early stages of the execution.

To reduce the profiling overhead, we store only the IDs of the executed paths and the identifiers of the accessed objects in thread-local buffers. Upon instrumentation, we associate information that is statically available at compile time (and hence remains constant among executions of the same path) to paths. For example, we associate to a path all IR instructions contained in the path, and for each IR instruction we store its corresponding source method. Our profiler implements two buffer-dumping modes. In the first mode, we dump the thread-local buffers when full, immediately

before storing a path that would not fit into the buffer, and upon thread termination. We use this mode when executing workloads for which we expect a normal termination. In the second mode, we memory map thread-local buffers to trace files. When a buffer is full, we remap the thread-local buffer to a different (higher) offset in the trace file. We use this mode when executing workloads for which we expect an abnormal termination (e.g., microservices workloads that are killed with a SIGKILL signal as explained later in Sec. 7.1). The reason is that, upon abnormal termination, threads may not execute the thread-termination handlers and hence may not dump buffers. Using memory-mapped files, the kernel ensures that traces are not lost.

To perform code ordering, we trace two different events depending on the proposed technique. For *cu ordering*, we trace *cu entry* events, while for *method ordering*, we trace *method entry* events. Finally, to perform heap-snapshot ordering, we trace all the identifiers of the accessed Java objects upon every field or array access. Since object identifiers represent runtime information, they are stored in the thread-local buffers together with the executed path IDs. When parsing a trace file, each path ID (associated with a fixed sequence of events) determines how many object identifiers are stored after the path ID.

6.2 Post-processing Analyses

To parse the traces and obtain ordering profiles, we implement a Java post-processing framework that implements ordering analyses. Each ordering analysis produces as output a CSV file that is used by Native Image. Ordering analyses are implemented as classes that exploit the visitor pattern and accept one event after the other in execution order. The framework reads the trace files, decodes the path IDs (i.e., obtains the sequence of events associated with the path ID and if present reads hashes stored after the path ID), and dispatches all the events occurring in the executed paths to the analyses. Each analysis internally keeps an ordered set that stores either the CUs, methods, or hashes in encounter order (and hence, in execution order). After the analyses have consumed all the executed paths/events, the ordered set of each analysis is dumped into a CSV file.

7 Evaluation

In this section, we first present our experimental setup (Sec. 7.1). Then, we present the page-fault reductions (Sec. 7.2) and execution-time speedups (Sec. 7.3) achieved by the proposed ordering strategies. Finally, we discuss the execution-time overhead of our tracing profiler (Sec. 7.4).

7.1 Evaluation Settings

We run our experiments on a machine equipped with a 16-core Intel Xeon Gold 6326 (2.90 GHz) and 256 GB of RAM running Linux Ubuntu (kernel v. 5.15.0-25-generic). Frequency scaling, turbo boost, hyper-threading, and address space randomization are disabled, CPU governor is set to "performance". We conduct our experiments on GraalVM Community Edition, based on OpenJDK 21, using the Graal compiler. We modify both Graal and the Native Image to implement our strategies. We perform our experiments on two different sets of benchmarks. To evaluate the improvements on the FaaS model, where there are no microservices or long-running server processes and any program that maps the input to the output can be a function, we employ AWFY [33]. The benchmark suite consists of 14 benchmarks designed to compare language implementations and optimize their compilers. To evaluate the improvements on microservices, we employ a `helloworld` workload implemented using three widely-used microservice frameworks: *micronaut* [34], *quarkus* [47], and *spring* [53]. We use *helloworld* because we want to measure the improvements in the startup of the microservice frameworks and not in the user application, which we evaluate using AWFY. While AWFY benchmarks are single threaded, microservice workloads are multi threaded. To compute the orderings in a multi-threaded setting, we concatenate the orderings of all the threads in thread creation order and remove duplicated entries. We build statically linked executables as recommended [30]. Since our goal is optimizing and evaluating the first binary execution, where data is not already present in RAM and needs to be fetched, we drop clean caches, as well as reclaimable slab objects such as *dentries* and *inodes* between benchmark iterations [52].

We execute our experiments employing a Solid-state Drive (SSD) and page size of 4 KB. We note that we executed the same experiments employing an NFS and obtained similar results. We do not employ AWS Lambda [3] or other cloud computing services [41] to perform our evaluation because these services do not allow collecting performance numbers by adequately customizing the evaluation settings. For example, they do not allow dropping caches between workload executions.

For each strategy (including the unmodified baseline), we build 10 native images for each benchmark. For each of these builds, we run 10 iterations to measure page faults and another 10 iterations to measure the end-to-end execution time and the elapsed time until the first response for AWFY and microservice workloads, respectively. End-to-end execution time measurements are obtained using `perf` [28]. To obtain the elapsed time until the first response, we start the time measurement, we run the microservice workload, and we continuously ping the endpoint exposed by the microservice. As soon as the workload sends back a response, we stop the time measurement and send a SIGKILL signal to kill the workload. We note that this evaluation setting is subject to measurement variability due to this inter-process communication. To determine page-fault reductions related to the `.text` and `.svm_heap` sections, we trace page faults using `perf` and extract only the page faults directed to the offsets

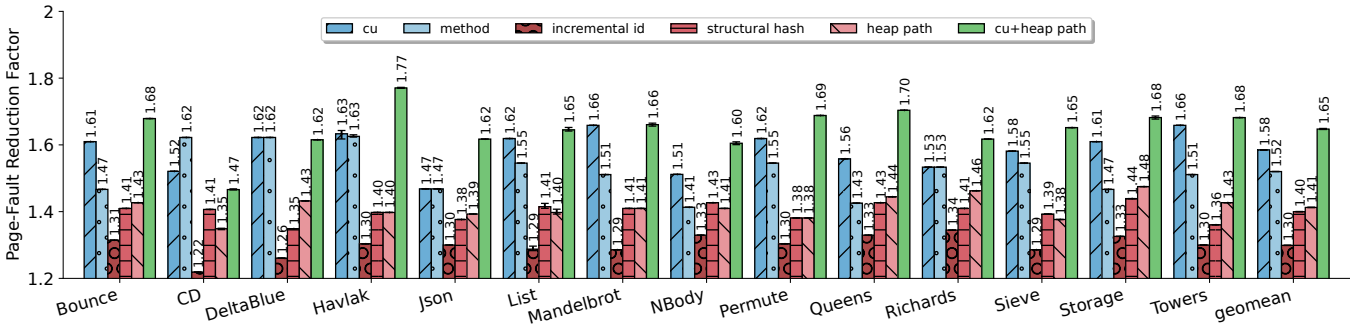


Figure 2. Page fault reduction achieved by the proposed ordering strategies on AWFY.

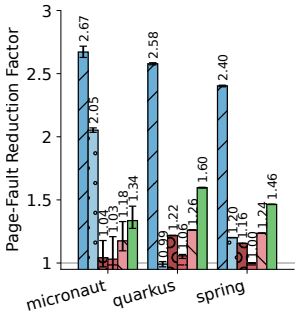


Figure 3. Page fault reduction achieved by the proposed ordering strategies on microservices.

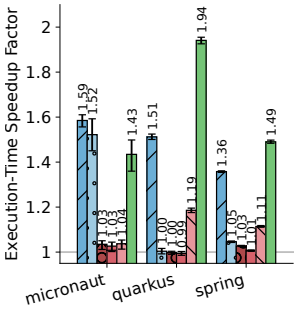


Figure 4. Execution-time speedup achieved by the proposed ordering strategies on microservices.

of the binaries belonging to these sections. In both cases, we compute the average of all the measurements.

All the figures shown in this section (except those in Sec. 7.4) report factors computed as $M_{baseline}/M_{optimized}$, where $M_{baseline}$ refers to the average measurement obtained without using our strategies and $M_{optimized}$ refers to the average measurement obtained using one of our strategies (higher is better). The benchmarks are reported on the x-axis of the plot, while the factors are reported on the y-axis. After the AWFY benchmarks, we report the geometric mean across all benchmarks. Above each bar, we report the exact factor. The black error bars represent 95% confidence intervals (CI) of the measurements.

For the *structural hash* strategy, we set `MAX_DEPTH` to 2, experimentally determined as a good trade-off between computational time, hash collision probability, and identity-matching probability across compilations. For code-ordering strategies, we report the page-fault reduction factors computed by considering only the page faults caused by the `.text` section of the binary, while for heap ordering strategies, we report the page-fault reduction factors computed by considering only the page faults caused by the `.svm_heap` section of the binary. To evaluate the combined benefits of the code- and heap-ordering strategies, we report both the page fault reductions and the execution-time speedups for a strategy named *cu+heap path*. In this strategy, we order both code and objects by combining the *cu* and *heap path* strategies, i.e., the code- and heap-ordering strategies, respectively, that yield to the highest reduction of page faults according to our experiments.

7.2 Page-Fault Reduction

Fig. 2 and 3 report the page-fault reductions obtained by the proposed ordering strategies.

Code-ordering strategies. Experimental results show that the *cu ordering* and *method ordering* strategies are both effective in reducing page faults related to the `.text` section of the binary for all the evaluated benchmarks. However, *cu ordering* outperforms *method ordering*, as on average it reduces page faults by 1.58× on AWFY and by 2.55× on microservices (while *method ordering* leads to factors of 1.52× on SSD and 1.35× on microservices). The maximum page fault reduction is achieved by *cu ordering* on the *Mandelbrot* and *Towers* AWFY benchmarks (1.66×), and on the *micronaut* microservice benchmark (2.67×).

Heap-ordering strategies. As the figures show, the *incremental id*, *structural hash*, and *heap path* ordering strategies introduce no page-fault increase on any benchmark (except *incremental id* on *quarkus* with factor 0.99×). On AWFY, while the average reductions of page faults related to the `.svm_heap` section of the binary is similar for *structural hash* and *heap path* (average of 1.40× and 1.41×, respectively), *incremental id* is less effective (average of 1.30×). On microservices, *heap path* (average of 1.22×) outperforms *incremental id* and *structural hash* (average of 1.14× and 1.03×, respectively). Experimental results indicate that, despite the segregation by type, one cannot rely on the encounter order when traversing the heap object graph. Instead, hashing the heap paths from the roots to the objects included in the heap snapshot is more robust.

We note that the evaluated benchmarks access a small percentage of the objects stored in the `.svm_heap` section

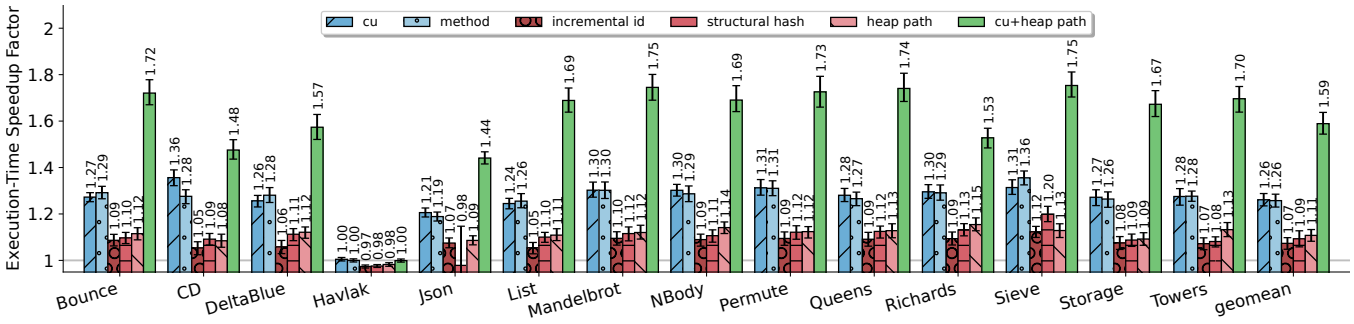


Figure 5. Execution-time speedup achieved by the proposed ordering strategies on AWFY.

of the binary (on average 4% on AWFY) and hence heap-ordering strategies need to be rather precise. Indeed, the heap snapshot does not only contain the user-allocated objects but also many `String` literals, `Class` instances, metadata `byte` arrays, and maps that dominate the size. The maximum page-fault reduction factor is achieved by *heap path* on *Storage* (1.48×) and *quarkus* (1.26×) on AWFY and microservices, respectively.

Combining Code- and Heap-ordering. When used together, the *cu* and the *heap path* ordering strategies introduce average page-fault reduction factors (related to both the `.text` and `.svm_heap` sections of the binary) of 1.65× and 1.46× on AWFY and microservices, respectively.

7.3 Execution-Time Speedup

In this section, we report the execution-time speedups introduced by the proposed ordering strategies. Fig. 4 and 5 show the speedup achieved by the code- and heap-ordering strategies separately, as well as their combined speedups.

On AWFY, both code-ordering strategies (*method* and *cu*) introduce an average speedup of 1.26×, while the *incremental id*, *structural hash*, and *heap path* ordering strategies introduce speedups of 1.07×, 1.09×, and 1.11×, respectively. On microservices, the strategies introduce an average speedup of 1.48× (*cu*), 1.17× (*method*), 1.02× (*incremental id*), 1.01× (*structural hash*), and 1.11× (*heap path*). Experimental results show that, on the evaluated benchmarks, code-ordering strategies achieve more speedups than heap-ordering ones. When combined, the *cu* and the *heap path* strategies introduce speedups of 1.59× on AWFY and 1.61× on microservices. While code-ordering strategies do not introduce slowdowns in any benchmark, heap-ordering strategies introduce minor slowdowns (0.97×–0.99×) on benchmarks *Havlak* and *quarkus*.

7.4 Profiling Overhead

Our tracing profiler incurs moderate overhead on the evaluated benchmarks. On AWFY, when employing the first buffer-dumping mode, the execution-time overhead for code ordering strategies *cu* and *method* is on average 1.21× and 1.83×, respectively, while the average execution-time overhead for heap-ordering strategies is 1.36×. On microservice benchmarks, when employing the second buffer-dumping mode (i.e., memory-mapped files), the execution-time overhead is 1.90×, 3.68×, and 2.16× for *cu*, *method*, and heap-ordering strategies, respectively. We note that the emitted instrumentation code is the same for all the different heap-ordering strategies. Hence, we report a single overhead factor associated with *incremental id*, *structural hash*, and *heap path*. We note that a high overhead introduced by our tracing profiler is a minor drawback since the observed workloads need to be profiled only once to obtain the profiles used to perform our ordering strategies and hence produce the optimized binary file. Moreover, since we observe and optimize short-running workloads typically executed on cloud environments, even a high overhead factor leads to a moderate total running time that does not impair the applicability of our approach.

8 Related Work

While our work focuses on the optimization of startup performance by improving low-level metrics, related work tackles startup performance improvements mostly by proposing techniques at different abstract levels, focusing either on the optimization of virtual machines, interpreter, and JIT compilers or on the optimization of Serverless platforms and functions. Instead, existing function- and heap-ordering approaches either do not aim at optimizing startup performance or are not suitable to Native Image, respectively. We discuss them in the following text.

Startup Performance Optimization of startup performance is a hot topic in the programming language community. Widely used virtual machines such as GraalVM [38] and the V8 JavaScript VM [18] implement techniques to pre-initialize the execution context [39, 57, 60]. Amazon Web Services Labs have recently announced LLRT (Low Latency Runtime), "a lightweight JavaScript runtime designed to address the growing demand for fast and efficient Serverless applications" [2]. Several techniques improve VM interpreter performance [6, 9, 45] and reduce the startup time of the JIT compiler [5, 32, 42, 59]. In contrast to such techniques, our

work focuses on a lower abstraction level, i.e., the reduction of I/O traffic (and hence page faults) during startup. The proposed ordering strategies are complementary to these techniques.

Serverless and FaaS Optimization Recent research focuses on the optimization of Serverless platforms and functions, as reported by a recent systematic review [55]. Techniques that optimize the cold start of the Serverless platform are intrinsically orthogonal to our approach and include, for example, instance prewarm preparation, function scheduling, and snapshot-based optimizations. The only approach we are aware of that optimizes cold-start performance of FaaS at the application level is FaaSLight [29], which reduces the code size of the application by separating code related to application functionalities from other code that can be loaded on-demand only when needed. Hence, FaaSLight has a different focus and is complementary to our approach.

Function Ordering Related work in the context of mobile applications reorders functions to reduce page faults and optimize startup time [23, 25] using PGO. However, these approaches do not focus on function inlining and divergences between compilations. Instead, they exploit profiles to modify compilation to reduce the binary size by performing function outlining. In Native Image, Graal's inlining is required to remove programming abstractions and produce performant binary code; outlining functions may potentially decrease performance and increase the number of objects in the binary file due to missed PEA optimizations. Hence, the above strategies do not work well in Native Image.

Differently from the proposed strategies, which focus on improving the performance of short-running applications, several techniques try to improve cache locality of long-running or large applications to speed up steady-state. The PH algorithm [44] implements a heuristic based on a weighted undirected dynamic call graph and is widely used by state-of-the-art compilers and tools. The C^3 algorithm [43] improves the PH algorithm by using a directed call graph instead of an undirected call graph. SARSA [11] is a reinforcement learning algorithm that reorders functions by exploiting a bidirectional function call graph. CodeMason [56] reorders function by performing binary profiling and rewriting. The GCC compiler offers several options to reorder functions and basic blocks in the object file upon linking time by using profiles or user-provided code annotations [15]. In practice, GCC places hot and unlikely functions into two distinct sections of the binary file named `.text.hot` and `.text.unlikely`, respectively, but does not optimize their ordering to reduce page faults.

Heap Ordering To the best of our knowledge, no previous work attempts to reorder objects in binary files to reduce page faults and improve startup time. Despite heap ordering is particularly relevant in Native Image (where the image heap occupies from 40% to 60% of the binary size), related work mostly proposes dynamic memory allocators to improve cache locality [10, 14, 19, 24], hence focusing only on runtime allocation. We note that some of these techniques exploit PGO. For instance, MaPHeA [37] collects heap allocation and access profiles to optimize the heap object management across all memory hierarchies. HALO [50] is a post-link PGO tool and runtime memory allocator that rearranges heap objects according to allocation profiles. Other work [20, 48, 54] focuses on improving cache locality by optimizing object data-layout. Finally, we are not aware of any prior work that accurately maps object identities across compilations or executions. Prior work exploits a time-based technique to align execution traces obtained from separate runs [21, 35, 36], possibly allowing performance analysis. Unfortunately, these approaches do not map the semantically same objects across compilations and hence may not be directly employed.

9 Concluding Remarks

In this paper, we propose a profile-guided methodology to reorder the layout of Native-Image binaries during compilation, with the goal of improving startup performance and locality. In particular, we propose two code-ordering strategies and three heap-ordering strategies, aiming at reducing page faults related to the code section and the heap-snapshot section of the binary, respectively. The heap-ordering strategies are based on a methodology (proposed in this paper) to match objects from a profile against the objects in the profile-guided build, which is necessary as object identities and the heap-snapshot contents are not persistent across Native-Image builds of the same program.

To perform the ordering strategies, we propose a profiling methodology to obtain the execution-order profile of methods and objects. We implement the ordering strategies in GraalVM Native Image and implement the profiling methodology in a tracing profiler within the Graal compiler. Finally, we evaluate the proposed code- and heap-ordering strategies, showing that they are effective in both reducing page faults and improving runtime performance, achieving an average page-fault reduction factor of 1.61× and an average execution-time speedup of 1.59×.

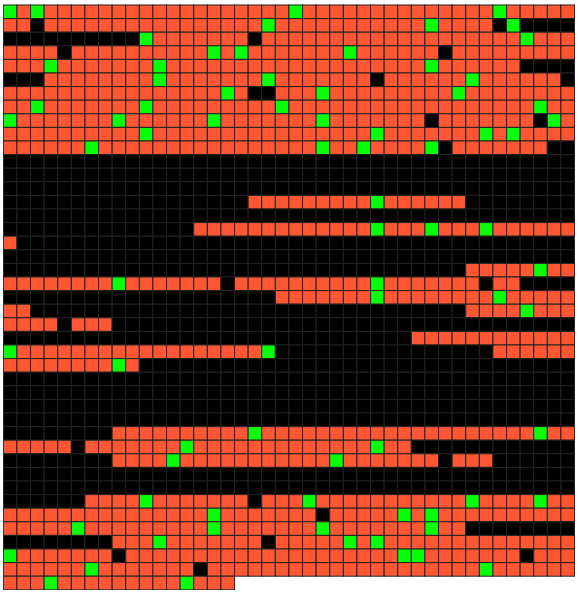
Acknowledgments

This work has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688). We thank the VM Research Group at Oracle Labs for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

A Appendix

In this appendix, we show a visual representation of the effects of our code orderings.

Visual Representation of Code-Ordering Effects. Fig. 6a and 6b show a visual representation of the page faults caused by the `.text` section occurring upon the execution of a regular binary and an optimized binary—produced by employing the *cu* strategy—of the AWFY *Bounce* workload, respectively. In the figures, each cell represents a page of the `.text` section of the binary files. Black cells represent physical pages in the binary file that are not mapped to virtual addresses of the process running the binary file. Green cells represent physical pages in the binary file that caused page faults. Red cells represent physical pages in the binary file that are mapped to virtual addresses of the process running the binary file but caused no page fault—these pages have been paged in by the operating system. Fig. 6a shows how page faults of a regular Native Image binary are distributed across the entire `.text` section. Instead, Fig. 6b shows how our technique reduces the number of page faults and compacts most of the executed methods in the first part of the `.text` section, showing the effectiveness of our ordering. We note that the executed methods which are placed at the end of the `.text` section are native methods included in statically-linked libraries. In our strategies, we do not profile and hence reorder native methods that are not compiled by the Graal compiler. We consider reordering these methods part of our future work. Furthermore, we plan to develop a similar visualization for the heap-snapshot section of the binary. This visualization may enable a fine-grained analysis of the included objects and a better understanding of the results reported in the paper, potentially enabling further improvements to our heap-ordering strategies.



(a) Regular binary

(b) Binary optimized by employing the *cu* strategy

Figure 6. Visual representation of the `.text` section of the AWFY benchmark *Bounce* and the page faults caused by this section.

B Artifact Appendix

The artifact [8] consists of a ready-to-use Docker image embedding our profiler as well as our modified GraalVM to generate optimized Native-Image binaries that reduce I/O traffic by changing their layout during compilation. A set of tools/scripts can be used to execute the benchmarks, collect, process, and plot page fault and performance measurements to replicate the evaluation presented in the paper. The artifact also contains the complete pre-collected measurements used to generated the original figures of the paper.

B.1 Overview

- Approximate time to install: 5 minutes (Sec. B.2.2).
- Approximate time to reproduce the figures of the paper using pre-collected data: 5 minutes (Sec. B.2.3).
- Approximate time to reproduce the results: minimum 5 hours (Sec. B.2.3).

B.2 Getting Started Guide

B.2.1 Requirements.

Hardware Requirements. Host machines should have at least 16GB of RAM, at least 30GB of free space, bash script support, and an Internet connection.

Software Requirements. Host machines should have Docker installed (we have tested Docker version `27.1.1, build 6312585` on Ubuntu 22.04 LTS). We have tested our bash scripts on Ubuntu 22.04 LTS. The bash scripts may not work on other operating systems.

B.2.2 Installing the Artifact.

1. Extract the artifact `tgz` available at the following URL: https://doi.org/10.5281/zenodo.13302630. It includes the ready-to-use evaluation Docker image, a script to run the artifact, and an extended version of this artifact appendix as a `README` file.
2. Verify the installation by executing the `run.sh` specifying `verify` as argument: `./run.sh verify`. If the artifact is correctly installed, you should see the following message to the standard output: `The artifact is correctly installed`.

Note: We require `sudo` privileges to drop clean caches between benchmark executions.

Note: This command tries to automatically install the kernel-specific packages to run `perf` on the host machine.

B.2.3 Using the Artifact.

The main script to reproduce the evaluation of our paper is `run.sh`, which receives one of the following arguments specifying the run mode:

1. `precollected`: Generates the figures shown in the paper using pre-collected data.
2. `evaluation`: Runs the benchmarks in the container and then generates the figures from the collected data.
3. `clean`: Cleans the environment by deleting the generated figures and the newly collected data.

Generating figures with pre-collected data. To generate the figures using pre-collected data (i.e., the same data used to generate the figures that appear in the paper), follow the instructions below:

1. Execute `run.sh` specifying `precollected` as argument: `./run.sh precollected`
2. The generated figures will be saved in the `output/figures-paper` folder in the host machine.

Generating figures with newly collected data. To generate the figures using newly collected data (i.e., to profile and execute the benchmarks and then generate the figures), follow the instructions below:

1. Execute `run.sh` specifying `evaluation` as argument: `./run.sh evaluation`
2. After the measurements are completed, the generated figures will be saved in the `output/figures` folder in the host machine. The raw data will be saved into the `output/data` folder, instead. The figures will be generated only after the execution of all the benchmarks.

Cleaning the Environment. To delete the generated figures and the newly collected data, follow the instructions below:

1. Execute `run.sh` specifying `clean` as argument: `./run.sh clean`

B.3 Overview of Claims

To reproduce the claims, we report the list of figures that should be reproduced. For each figure in the paper, we report a brief description and the list of generated PDFs to be checked when executing both the `precollected` and `evaluation` command.

The `precollected` command generates Figures 2 and 5 shown in the paper. These figures are saved in folder `output/figures-paper`.

The `evaluation` command generates the figures using newly collected data. These figures are saved in folder `output/figures`.

With the goal of easing the evaluation of the artifact, we ported our evaluation to a containerized environment. Nonetheless, the use of containerization may significantly impact our measurements, particularly those based on execution time. Moreover, different host machines with different hardware capabilities may yield different execution times. We note that the performance measurements used to generate the figures in the paper have been collected in an isolated environment with minimal perturbation where (almost) no other process was being executed. Newly collected data may lead to the generation of figures with different numbers w.r.t. those shown in the paper. However, we expect the data trends and the figures generated with newly collected data to be similar to the ones shown in the paper, as detailed below.

Note: We recommend disabling frequency scaling, turbo boost, hyper-threading, and address space randomization. Moreover, we recommend setting CPU governor to *performance* (Sec. 7.1). Please find the commands at following URL: https://llvm.org/docs/Benchmarking.html.

B.3.1 Page-Fault Reduction (Figure 2).

In Figure 2, we report the page-fault reductions obtained by the proposed ordering strategies.

Precollected Data.

- Figure 2 in the paper should be compared with the generated figure at `output/figures-paper/ssd_pagefault-reductions.pdf`

Newly Collected Data.

- The generated figure is saved at `output/figures/pagefault-reductions.pdf`.
- We expect the page-fault reductions of strategy *cu+heap path* to be greater or similar to the page-fault reductions of strategies *cu* and *method*.

- We expect strategies *cu* and *method* to yield greater page-fault reductions than strategies *incremental id*, *structural hash*, and *heap path*.

B.3.2 Execution-Time Speedup (Figure 5). In Figure 5, we evaluate the speedup achieved by the proposed ordering strategies.

Precollected Data.

- Figure 5 in the paper should be compared with the generated figure at `output/figures-paper/ssd_speedups.pdf`

Newly Collected Data.

- The generated figure is saved at `output/figures/speedups.pdf`.
- We expect no slowdown, with the exception of benchmark *Havlak*.
- We expect strategies *cu* and *method* to yield greater speedups than strategies *incremental id*, *structural hash*, and *heap path*.
- We expect greater speedups for strategy *cu+heap path*.

B.4 Reusing and Modifying the Artifact

In the `README` file of the artifact, we report additional notes that are not needed for the "regular" artifact evaluation but help in reusing and extending the artifact. These notes are not needed for reproducing the evaluation of our paper. We do not report these notes in this appendix due to a matter of space.

B.5 Troubleshooting

- If the evaluation fails with an unexpected error, try running the `run.sh` script with `sudo`. The reason is that our evaluation scripts drop clean caches between benchmark iterations (Sec. 7.1).
- If you have disk space issues with docker, try running `docker system prune -a` or restart the docker service.

References

[1] Adam Horvath. 2012. MurMurHash3, An Ultra Fast Hash Algorithm for C# / .NET. https://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html

[2] Amazon Web Services - Labs. 2024. LLRT GitHub Repository. https://github.com/awslabs/llrt

[3] Amazon Web Services, Inc. or its affiliates. 2024. AWS Lambda. https://aws.amazon.com/lambda/

[4] Amazon Web Services, Inc. or its affiliates. 2024. Lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

[5] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *OOPSLA*. 297–311. https://doi.org/10.1145/1094811.1094835

[6] Matteo Basso, Daniele Bonetta, and Walter Binder. 2023. Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance. In *GPCE*. 1–13. https://doi.org/10.1145/3624007.3624050

[7] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2023. Optimization-Aware Compiler-Level Event Profiling. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 10 (Jun 2023), 50 pages. https://doi.org/10.1145/3591473

[8] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2024. Artifact associated to the paper "Improving Native-Image Startup Performance" published in CGO'25. https://doi.org/10.5281/zenodo.13760307 artifact.

[9] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (Jun 1973), 370–372. https://doi.org/10.1145/362248.362270

[10] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS*. 139–149. https://doi.org/10.1145/291006.291036

[11] Weibin Chen and Yeh-Ching Chung. 2022. Profile-Guided Optimization for Function Reordering: A Reinforcement Learning Approach. In *SMC*. 2326–2333. https://doi.org/10.1109/SMC53654.2022.9945280

[12] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation *(APPLC'13)*. 1–9.

[13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. 1–10. https://doi.org/10.1145/2542142.2542143

[14] Yi Feng and Emery D. Berger. 2005. A Locality-improving Dynamic Memory Allocator. In *MSP*. 68–77. https://doi.org/10.1145/1111583.1111594

[15] Free Software Foundation. 2024. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[16] Free Software Foundation. 2024. Program Instrumentation Options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[17] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS*. 386–400. https://doi.org/10.1145/3445814.3446757

[18] Google. 2024. V8 JavaScript Engine. https://www.v8.dev

[19] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *PLDI*. 177–186. https://doi.org/10.1145/173262.155107

[20] Christopher Haine, Olivier Aumage, and Denis Barthou. 2017. Rewriting System for Profile-Guided Data Layout Transformations on Binaries. In *Euro-Par*. 260–272. https://doi.org/10.1007/978-3-319-64203-1_19

[21] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. 2005. Automating Vertical Profiling. In *OOPSLA*. 281–296. https://doi.org/10.1145/1094811.1094834

[22] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *PASTE*. 54–61.

[23] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing Function Layout for Mobile Applications. In *LCTES*. 52–63. https://doi.org/10.1145/3589610.3596277

[24] Alin Jula and Lawrence Rauchwerger. 2009. Two Memory Allocators that Use Hints to Improve Locality. In *ISMM*. 109–118. https://doi.org/10.1145/1542431.1542447

[25] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-guided Size Optimization for Native Mobile Applications. In *CC*. 243–253. https://doi.org/10.1145/3497776.3517764

[26] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *ManLang*. 1–13. https://doi.org/10.1145/3237009.3237013

[27] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*. 126–137. https://doi.org/10.1145/3168811

[28] Linus Torvalds. 2024. Linux perf GitHub Repository. https://github.com/torvalds/linux/tree/master/tools/perf

[29] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 119 (Jul 2023), 29 pages. https://doi.org/10.1145/3585007

[30] LLVM Project. 2024. Benchmarking Tips. https://llvm.org/docs/Benchmarking.html

[31] LLVM Project. 2024. How To Build Clang and LLVM with Profile-Guided Optimizations. https://llvm.org/docs/HowToBuildWithPGO.html

[32] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *ManLang*. 105–118. https://doi.org/10.1145/3132190.3132210

[33] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *DLS*. 120–131. https://doi.org/10.1145/2989225.2989232

[34] Micronaut Foundation. 2024. Micronaut Framework. https://micronaut.io/

[35] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2006. Aligning Traces for Performance Evaluation. In *IPDPS*. 291–298. https://doi.org/10.1109/IPDPS.2006.1639592

[36] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. 2007. Time Interpolation: So Many Metrics, So Few Registers. In *MICRO*. 286–300. https://doi.org/10.1109/MICRO.2007.27

[37] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W. Lee, Jung Ho Ahn, and Eojin Lee. 2022. MaPHeA: A Framework for Lightweight Memory Hierarchy-aware Profile-guided Heap Allocation. 22, 1, Article 2 (Dec 2022), 28 pages. https://doi.org/10.1145/3527853

[38] Oracle and/or its affiliates. 2021. GraalVM. https://www.graalvm.org

[39] Oracle and/or its affiliates. 2021. GraalVM: Native Image. https://www.graalvm.org/jdk21/reference-manual/native-image/

[40] Oracle and/or its affiliates. 2024. Class String. https://docs.oracle.com/en/java/javase/21/docs//api/java.base/java/lang/String.html#intern()

[41] Oracle and/or its affiliates. 2024. Cloud Functions. https://www.oracle.com/cloud/cloud-native/functions/

[42] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *CGO*. 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

[43] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-scale Data-center Applications. In *CGO*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[44] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *PLDI*. 16–27. https://doi.org/10.1145/93542.93550

[45] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *POPL*. 322–332. https://doi.org/10.1145/199448.199526

[46] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-In-Time Compilers. In *CGO*. 164–179. https://doi.org/10.1109/CGO.2019.8661171

[47] Red Hat. 2024. Quarkus. https://quarkus.io/

[48] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An Efficient Profile-analysis Framework for Data-layout Optimizations. In *POPL*. 140–153. https://doi.org/10.1145/565816.503287

[49] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *CC*. 126–137. https://doi.org/10.1007/3-540-36579-6_10

[50] Joe Savage and Timothy M. Jones. 2020. HALO: Post-link Heap-layout Optimisation. In *CGO*. 94–106. https://doi.org/10.1145/3368826.3377914

[51] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *CGO*. 165–174. https://doi.org/10.1145/2581122.2544157

[52] The kernel development community. 2024. Documentation for /proc/sys/vm/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html?highlight=drop_caches#drop-caches

[53] VMware Tanzu. 2024. Spring Framework. https://spring.io/

[54] Yongliang Wang, Naijie Gu, Junjie Su, Dongsheng Qi, and Zhuorui Ning. 2022. Data Layout Optimization based on the Spatio-Temporal Model of Field Access. In *AEMCSE*. 238–244. https://doi.org/10.1109/AEMCSE55572.2022.00055

[55] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 131 (Jul 2023), 61 pages. https://doi.org/10.1145/3579643

[56] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *FEAST*. 47–53. https://doi.org/10.1145/3338502.3359763

[57] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter Bernard Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29. https://doi.org/10.1145/3360610

[58] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. In *PLDI*. 24 pages. https://doi.org/10.1145/3656417

[59] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 124 (Oct 2018), 23 pages. https://doi.org/10.1145/3276494

[60] Yang Guo. 2015. Custom Startup Snapshots. https://www.v8.dev

Received 2024-05-30; accepted 2024-07-22