Improving Native-Image Startup Performance

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

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

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

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

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 [3, 40]. 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 1.65× when using a Solid-state Drive (SSD), and of 1.68× when using Network File System (NFS). This reduction results in an average execution-time speedup of 1.59× (SSD) and 1.58× (NFS).

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. 2024. Improving Native-Image Startup Performance. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [52], 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 [16], 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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [52], a few techniques try to perform startup optimizations at the application level. For instance, GraalVM Native Image [55] 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 GraalVM 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). Then, 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 easily 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. 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), so 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 compilations 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?" benchmark suite [32] (Sec. 7), showing that they are effective in both reducing page faults and improving runtime performance when employing both a Solid-state Drive (SSD) and a NFS, leading to an average page-fault reduction factor of 1.65× (SSD) and 1.68× (NFS) and an average execution-time speedup of 1.59× and 1.58× (NFS).

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 (Sec. 2.1), heap snapshotting (Sec. 2.2), and profile-guided optimizations (Sec. 2.3).

2.1 Ahead-of-time (AOT) Compilation

To reduce the startup time of Java workloads, GraalVM Native Image [38, 55] (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 [12], an optimizing compiler that performs inlining [45], escape-analysis [49], and various other optimizations [25, 26]. 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 [21, 21, 47, 54, 55], 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 [54]. 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.

2.2 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) [49], 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.

2.3 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 [30] and GCC [15], 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* [21, 21, 47, 54, 55] 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

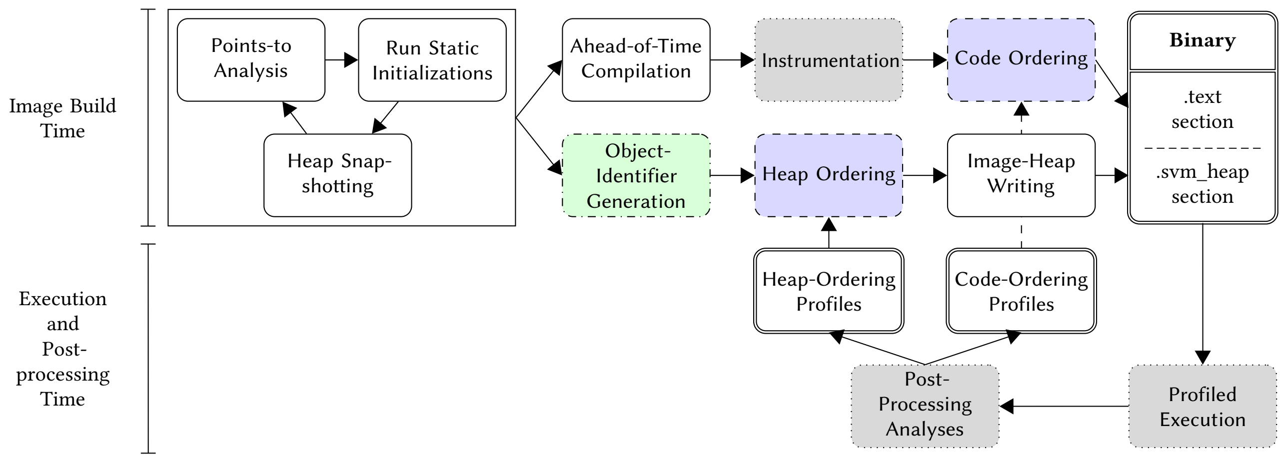


Figure 1. Integration of the proposed binary-reordering methodology into the Native-Image building process.

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:

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(); c();` in the case where the conditions of all the `if` statements evaluate to `true`. This strategy produces the CU-invocation trace `a, c, 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, c`, leading to the ordering `a, b, c`.

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

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
```

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
bytes ← encodeToBytes(entity, 0)
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
bytes ← newByteBuffer()
if entity.isNull() then
    bytes.append(0)
    return bytes
bytes.append(entity.type().fullyQualifiedName())
shouldRecurse ← depth < MAX_DEPTH
if entity.isPrimitive() or entity.isString() then
    bytes.append(entity.value())
else if entity.isObjectInstance() then
    fields ← entity.fields()
    for k ← 1 to fields.length() do
        field ← entity.getFieldWrapper(k)
        if shouldRecurse or field.isPrimitive() or
          field.isString() then
            bytes.append(field.type().fullyQualifiedName())
            bytes.append(encodeToBytes(field, depth +
              1))
else if entity.isArray() then
    elementType ← entity.elementType()
    bytes.append(elementType.fullyQualifiedName())
    bytes.append(entity.length())
    if shouldRecurse or elementType.isPrimitive() or
      elementType.isString() then
        for k ← 1 to entity.length() do
            bytes.append(k)
            element ← entity.getElementWrapper(k)
            bytes.append(encodeToBytes(element, depth+
              1))
return bytes
```

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 (profile-driven) 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 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 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 (line 14), as well as the bytes resulting from a recursive call that takes the field entity and the depth (incremented by 1) as parameters (line 15).

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 16–19). 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 20–24). The algorithm terminates by returning the byte buffer (line 25).

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 [39]), "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.

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)
```

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) [11, 12] 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. We dump the thread-local buffers when full, immediately before storing a path that would not fit into the buffer, and upon thread termination.

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 discuss the page-fault reductions (Sec. 7.2) and execution-time speedups (Sec. 7.3) achieved by the proposed ordering strategies.

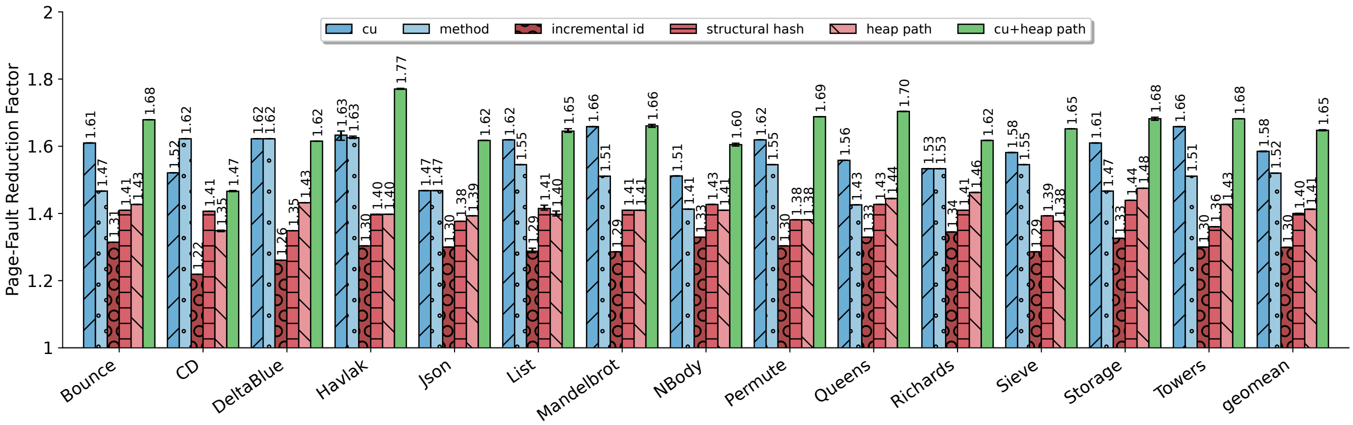
7.1 Evaluation Settings

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 "Are We Fast Yet?" [32], a benchmark suite consisting of 14 benchmarks specifically designed to compare language implementations and optimize their compilers. We build statically linked executables as recommended [29]. 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 [50].

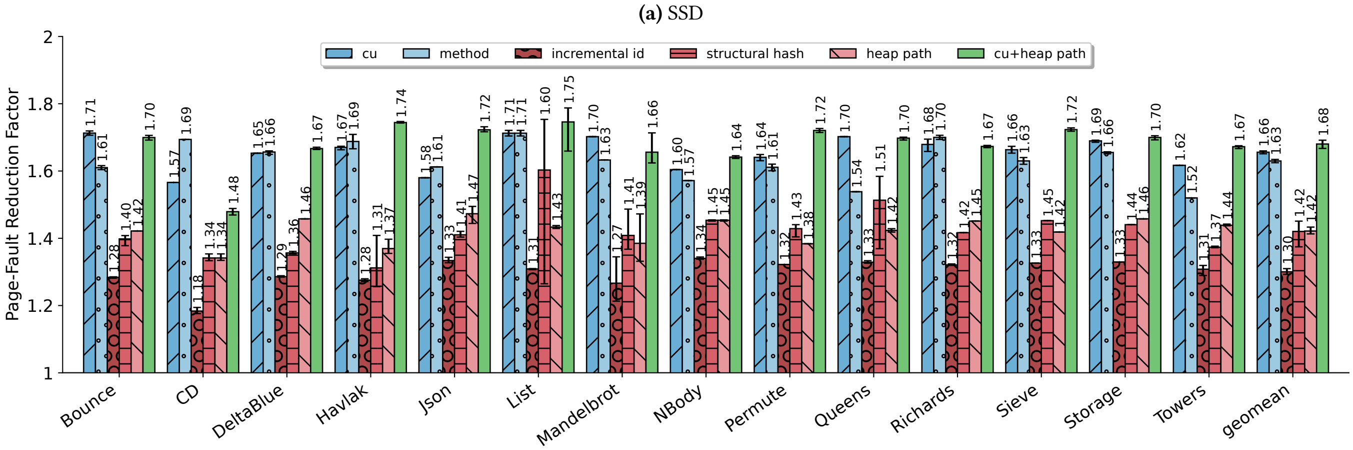
We execute our experiments employing both an SSD and a NFS. This allows us to consider both systems where the executable is fully downloaded before the first execution and systems that employ NFS. When using the SSD, the page size is 4 KB. The Linux NFS client runs NFS v. 4.2 and is configured with the `rsize` and `wsize` parameters [33] equal to 1 MB, i.e., 1 MB is the maximum number of bytes in each network read/write request. We note that 1 MB is the maximum value supported by the Linux NFS client, allowing one to limit the number of network requests (and hence context switches), thus minimizing the performance impact of page faults. As a consequence, the reduction of page fault and execution-time speedup reported in the paper can be seen as a lower bound—they may be larger in systems with lower `rsize` and `wsize` values. We do not employ AWS Lambda [3] or other cloud computing services 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 end-to-end execution time and another 10 iterations to measure page faults. End-to-end execution time measurements, often used by Serverless platforms as performance metrics, are obtained using `perf` [27]. 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 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 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



(a) SSD



(b) NFS

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

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 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.

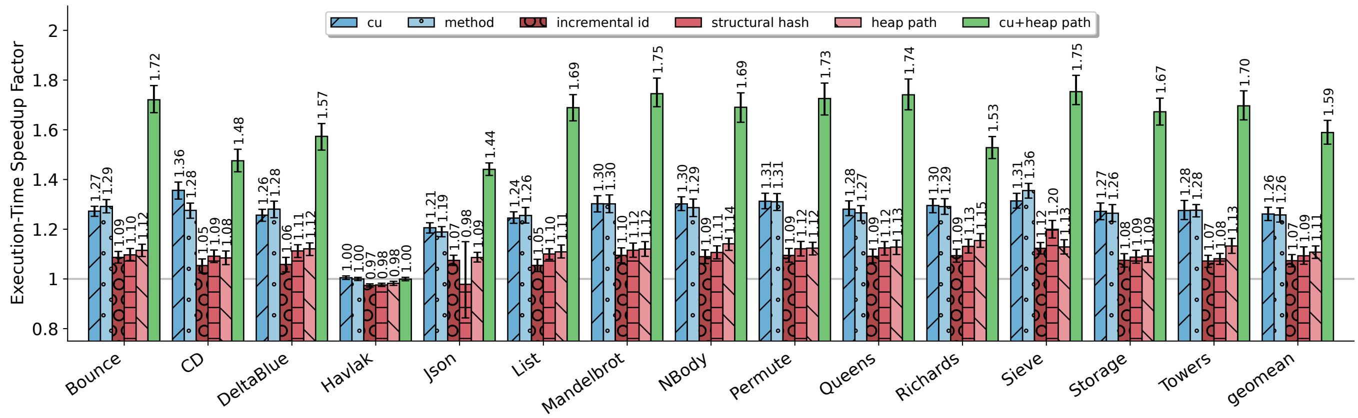
For the *structural hash* strategy, we set `MAX_DEPTH` to 2, experimentally determined as a good trade-off between computational time, hash collision probability, and identity-matching probability across compilations. For code-ordering strategies, we report the page-fault reduction factors computed by 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

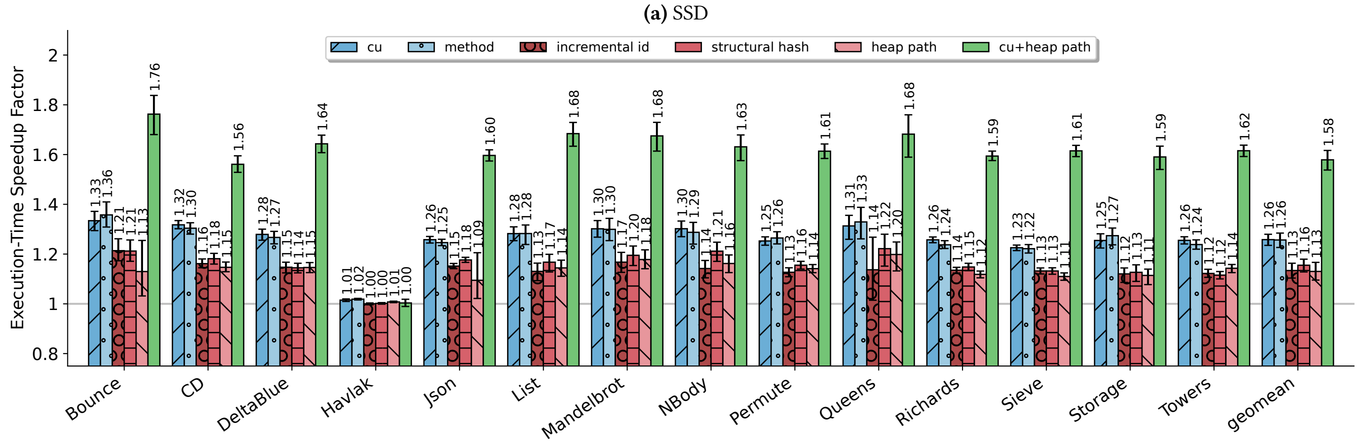
Fig. 2a and 2b report the page-fault reductions obtained by the proposed ordering strategies.

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 SSD and by 1.66× on NFS (while *method ordering* leads to factors of 1.52× on SSD and 1.63× on NFS). The maximum page fault reduction is achieved by *cu ordering* on the *Mandelbrot* and *Towers* benchmarks (1.66×, on SSD), and on the *Bounce* and *List* benchmarks (1.71×, on NFS).

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. While the average reductions of page faults related to the `.svm_heap` section of the binary is similar for *structural hash* and *heap*



(a) SSD



(b) NFS

Figure 3. Execution-time speedup achieved by the proposed ordering strategies.

path (average of 1.40× and 1.41× on SSD, respectively, and of 1.42× for both strategies on NFS), *incremental id* is less effective (average of 1.30× on both SSD and NFS). This indicates 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 on average 4% of the objects stored in the `.svm_heap` section of the binary 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 on SSD is achieved by *heap path* on *Storage* (1.48×) while on NFS by *structural hash* on *List* (1.60×).

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 1.68× on SSD and NFS, respectively, indicating that code and object reordering are synergistic.

7.3 Execution-Time Speedup

In this section, we report the execution-time speedups introduced by the proposed ordering strategies. Fig. 3a and 3b show the speedup achieved by the code- and heap-ordering strategies separately, as well as their combined speedups, on both SSD and NFS.

On SSD, 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 NFS, the strategies introduce an average speedup of 1.26× (*cu* and *method*), 1.13× (*incremental id* and *heap path*) and 1.16× (*structural hash*). 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 SSD and 1.58× on NFS. While code-ordering strategies do not introduce slowdowns in any benchmark, heap-ordering strategies introduce minor slowdowns (0.97×–0.98×) on the long-running benchmark *Havlak*, where steady-state performance is more important than startup performance.

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 (Sec. 8.1) or on the optimization of Serverless platforms and functions (Sec. 8.2). Instead, existing function- (Sec. 8.3) and heap-ordering (Sec. 8.4) approaches either do not aim at optimizing startup performance or are not suitable to Native Image, respectively.

8.1 Startup Performance

Optimization of startup performance is a hot topic in the programming language community. Widely used virtual machines such as GraalVM [37] and the V8 JavaScript VM [17] implement techniques to pre-initialize the execution context [38, 55, 57]. 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, 8, 44] and reduce the startup time of the JIT compiler [5, 31, 41, 56]. 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.

8.2 Serverless and FaaS Optimization

Recent research focuses on the optimization of Serverless platforms and functions, as reported by a recent systematic review [52]. 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 [28], 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.

8.3 Function Ordering

Related work in the context of mobile applications reorders functions to reduce page faults and optimize startup time [22, 24] 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 [43] 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 [42] improves the PH algorithm by using a directed call graph instead of an undirected call graph. SARSA [10] is a reinforcement learning algorithm that reorders functions by exploiting a bidirectional function call graph. CodeMason [53] 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 [14]. 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.

8.4 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 [9, 13, 18, 23], hence focusing only on runtime allocation. We note that some of these techniques exploit PGO. For instance, MaPHeA [36] collects heap allocation and access profiles to optimize the heap object management across all memory hierarchies. HALO [48] is a post-link PGO tool and runtime memory allocator that rearranges heap objects according to allocation profiles. Other work [19, 46, 51] 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 [20, 34, 35], 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 1.65× when using a SSD, of 1.68× when using NFS, and an average execution-time speedup of 1.59× (SSD) and 1.58× (NFS).

We plan to submit a publicly available artifact to the artifact evaluation process.

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.

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.

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

[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.

[8] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (Jun 1973), 370–372.

[9] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS*. 139–149.

[10] Weibin Chen and Yeh-Ching Chung. 2022. Profile-Guided Optimization for Function Reordering: A Reinforcement Learning Approach. In *SMC*. 2326–2333.

[11] 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.

[12] 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.

[13] Yi Feng and Emery D. Berger. 2005. A Locality-improving Dynamic Memory Allocator. In *MSP*. 68–77.

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

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

[16] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS*. 386–400.

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

[18] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *PLDI*. 177–186.

[19] Christopher Haine, Olivier Aumage, and Denis Barthou. 2017. Rewriting System for Profile-Guided Data Layout Transformations on Binaries. In *Euro-Par*. 260–272.

[20] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. 2005. Automating Vertical Profiling. In *OOPSLA*. 281–296.

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

[22] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing Function Layout for Mobile Applications. In *LCTES*. 52–63.

[23] Alin Jula and Lawrence Rauchwerger. 2009. Two Memory Allocators that Use Hints to Improve Locality. In *ISMM*. 109–118.

[24] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-guided Size Optimization for Native Mobile Applications. In *CC*. 243–253.

[25] 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.

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *DLS*. 120–131.

[33] Michael Kerrisk. 2024. nfs(5) – Linux Manual Page. https://www.man7.org/linux/man-pages/man5/nfs.5.html

[34] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2006. Aligning Traces for Performance Evaluation. In *IPDPS*. 291–298.

[35] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. 2007. Time Interpolation: So Many Metrics, So Few Registers. In *MICRO*. 286–300.

[36] 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.
[37] Oracle and/or its affiliates. 2021. GraalVM. https://www.graalvm.org
[38] Oracle and/or its affiliates. 2021. GraalVM: Native Image. https://www.graalvm.org/jdk21/reference-manual/native-image/
[39] 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()
[40] Oracle and/or its affiliates. 2024. Cloud Functions. https://www.oracle.com/cloud/cloud-native/functions/
[41] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *CGO*. 340–350.
[42] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-scale Data-center Applications. In *CGO*. 233–244.
[43] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. *SIGPLAN Not.* 25, 6 (Jun 1990), 16–27.
[44] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *POPL*. 322–332.
[45] 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.
[46] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An Efficient Profile-analysis Framework for Data-layout Optimizations. In *POPL*. 140–153.
[47] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *CC*. 126–137.
[48] Joe Savage and Timothy M. Jones. 2020. HALO: Post-link Heap-layout Optimisation. In *CGO*. 94–106.
[49] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *CGO*. 165–174.
[50] 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
[51] 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.
[52] 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.
[53] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *FEAST*. 47–53.
[54] Christian Wimmer, Codrut Stancu, Kozak David, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. In *PLDI*. 24 pages.
[55] 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.
[56] 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.
[57] Yang Guo. 2015. Custom Startup Snapshots. https://www.v8.dev