Heap-Snapshot Matching and Ordering using CAHPs: A Context-Augmented Heap-Path Representation for Exact and Partial Path Matching using Prefix Trees

MATTEO BASSO, Università della Svizzera Italiana (USI), Switzerland
ALEKSANDAR PROKOPEC, Oracle Labs, Switzerland
ANDREA ROSÀ, Università della Svizzera Italiana (USI), Switzerland
WALTER BINDER, Università della Svizzera Italiana (USI), Switzerland

GraalVM Native Image is increasingly used to optimize the startup performance of applications that run on the Java Virtual Machine (JVM), and particularly of Function-as-a-Service and Serverless workloads. Native Image resorts to Ahead-of-Time (AOT) compilation to produce a binary from a JVM application that contains a snapshot of the pre-initialized heap memory, reducing the initialization time and hence improving startup performance. However, this performance improvement is hindered by page faults that occur when accessing objects in the heap snapshot. Related work has proposed profile-guided approaches to reduce page faults by reordering objects in the heap snapshot of an optimized binary based on the order in which objects are first accessed, obtaining this information by profiling an instrumented binary of the same application. This reordering is effective only if objects in the instrumented binary can be matched to the semantically equivalent ones in the optimized binary. Unfortunately, this is very challenging because objects do not have unique identities and the heap-snapshot contents are not persistent across Native-Image builds of the same program.

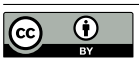
This work tackles the problem of matching heap snapshots, and proposes a novel approach to improve the mapping between semantically equivalent objects in different binaries of a Native-Image application. We introduce the concept of *context-augmented heap path (CAHP)*—a list of elements that describes a path to an object stored in the heap snapshot. Our approach associates a CAHP to each object in a way that is as unique as possible. Objects with the same CAHP across different binaries are considered semantically equivalent. Moreover, since some semantically equivalent objects may have different CAHPs in the instrumented and optimized binaries (due to nondeterminism in the image-build process and other factors), we present an approach that finds, for each unmatched CAHP in the optimized binary, the most similar CAHP in the instrumented binary, associating the two objects. We integrate our approach into Native Image, reordering the objects stored in the heap snapshot more efficiently using the improved mapping. Our experiments show that our approach leads to much less page faults (2.98× on average) and considerably improves startup time (1.98× on average) w.r.t. the original Native-Image implementation.

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

Additional Key Words and Phrases: GraalVM, Native Image, Startup Performance, Heap Path, Binary Reordering, Profiling, Profile-guided Optimizations, Serverless Computing, Function-as-a-Service.

ACM Reference Format:
Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2025. Heap-Snapshot Matching and Ordering using CAHPs: A Context-Augmented Heap-Path Representation for Exact and Partial Path Matching

Authors' Contact Information: 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.



ACM 2475-1421/2025/10-ART325
https://doi.org/10.1145/3763103

using Prefix Trees. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 325 (October 2025), 28 pages. https://doi.org/10.1145/3763103

1 Introduction

Serverless and Function-as-a-Service (FaaS) workloads are becoming prevalent in cloud runtimes and data centers. Unlike traditional server-side applications that run for a long time or perform extensive computations, these workloads are typically short-running, and therefore benefit less from optimizations performed by a Just-In-Time (JIT) compiler [1]. In these workloads, optimizing startup performance is often equally important as optimizing steady-state performance. Doing so is key to saving computational resources, maximizing the throughput of cloud services, and minimizing the chances of not meeting the service-level agreements (SLAs) of service providers.

A prominent approach to optimizing startup performance is to ahead-of-time (AOT) compile the code (e.g., application classes, standard-library classes, statically linked native code) to a native executable binary. Doing so reduces the time spent in initializing and compiling the runtime, and helps speed up application startup. In the context of Java Virtual Machine (JVM), this technique is implemented in Native Image [50], which creates a binary file from a JVM application,[1] preinitializing the Java environment at build time. A distinguishing feature of Native Image is that, in addition to the executable code, the generated binary also contains a snapshot of the preinitialized heap memory (i.e., Java objects).

Problem. Although embedding the code and a heap snapshot in the binary reduces the initialization time, a larger binary may still considerably impair the startup performance. Related work has shown that page faults[2] can significantly increase the startup time of Native-Image applications [4]. Page faults in Native-Image binaries can arise either from accesses to code or to the objects in the heap snapshot. In the binary, one page typically contains many accessed and unaccessed methods/objects. A promising approach to decrease page faults in Native-Image binaries is to rearrange code and objects in the binary, compacting them in fewer contiguous pages according to the order in which code is executed and objects are accessed. In particular, related work [4] achieves this owing to a profile-guided approach, which generates an instrumented binary of a Native-Image application to collect a profile that records the order in which code is first executed and in which objects are accessed[3]. This profile is then used to create a second binary in which the code and objects are laid out according to the order reflected in the profile.

Challenges. To be effective, the above approach requires that the code and objects in the profiles (obtained from the instrumented binary) can be matched to the corresponding ones in the optimized binary. While this is easy for code (because methods are unequivocally identified by their signatures), doing so for objects is very challenging, because an object does not have a unique name or identifier, and the heap-snapshot contents are not necessarily the same across image builds, due to nondeterminism during AOT compilation and during the execution of class initializers (§2.1). The aforementioned work attempts to address this challenge by hashing the structure and values in the objects and trying to reconcile semantically equivalent objects[4] by

[1] In this article, we refer to the binary files created by Native Image with both the terms "image" and "binary".

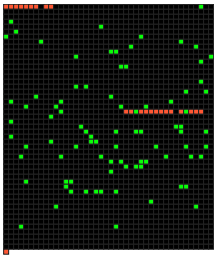
[2] A page fault is an exception that occurs when a process attempts to access code/data that is in its address space but is not currently loaded in memory. Upon the occurrence of a page fault, the OS loads the page (i.e., a fixed-length contiguous block of memory) containing the data from storage into memory. The page contains not only the data that will be accessed but also potentially other data that will not be accessed.

[3] Code and Image heap objects are always pinned to the same pages and placed at the same address in memory. Only the first access causes a page fault if the code/object is stored in a single page.

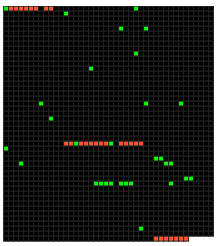
[4] In this article, we consider as *semantically equivalent* objects that have equivalent accesses in two execution traces of the program.

comparing the computed hashes. While prior work is effective in reducing page faults related to code accesses, the performance improvements due to object-reordering are minor in comparison.

Goal. This work addresses the object-matching problem with a novel approach that improves the mapping between objects in the instrumented binary and the semantically equivalent ones in the optimized binary. Our object-matching heuristics enable a better heap layout, i.e., a serialization of the heap object graph optimized for reducing page faults. Since our approach modifies only the serialized representation of the heap object graph and not the heap object graph itself, we do not impair the correctness of the binary (§5). Suboptimal layouting only leads to a lower reduction of page faults w.r.t. optimal layouting, as objects that are not matched or wrongly matched are likely to still cause page faults.



(a) Related work [4].



(b) Our strategy.

Fig. 1. Visual representation of page faults on the heap section of Native-Image binaries for workload *Bounce*.

Achievements. We show that our approach significantly reduces page faults related to object accesses and considerably improves the startup performance of Native-Image applications (§6). Fig. 1 illustrates how the new object-mapping approach reduces page faults. The figure shows the page faults that occurred in the heap section of the binary (i.e., when accessing objects) during the execution of the *Bounce* workload from the "Are We Fast Yet?" (AWFY) suite [29] (see §6). Fig. 1a refers to the binary produced by the most efficient object-ordering technique proposed in related work [4] (called HEAP PATH). Fig. 1b refers to the binary produced by the strategy presented in this article. Each cell represents a page in the binary. Black pages were not mapped to virtual addresses, green pages caused page faults, and red pages were paged-in (premapped) by the operating system. Red and (particularly) green cells are indicative of startup inefficiencies. Fig. 1a shows that page faults are spread throughout the heap, indicating the potential for a better layout—many objects are not matched and hence not compacted in fewer pages, still causing page faults. Fig. 1b shows fewer localized page faults, indicating that our approach maps and orders objects more effectively.

Contributions. This work makes the following contributions.

- **CAHP-based object matching.** We propose a novel approach to improve mappings between semantically equivalent objects in different binaries (§3 and §4). We define the concept of *context-augmented heap path (CAHP)*, i.e., a list of elements that describes each object in the heap. In the instrumented-image build, we compute the CAHPs of all objects. In the optimized build, we again compute the CAHP associated with each object in the heap of the optimized image. We then check whether the same CAHP is associated with an object from the instrumented-image. If a match is found, the objects associated with the matched CAHP are considered semantically equivalent (*exact path matching*). CAHPs effectively enable the matching of more objects w.r.t. existing strategies.
- **Partial path matching.** Unfortunately, due to divergences between the instrumented and the optimized builds (§2.1), semantically equivalent objects may have different CAHPs between builds, causing a mismatch. To mitigate these inaccuracies and to further reduce page faults, we propose a technique (called *partial path matching*) that identifies, for each unmatched CAHP in the optimized image, the most similar CAHP in the instrumented image, and associates the respective objects. In the optimized build, we use the object-equivalences computed by the exact and partial path matching to order the objects in the heap snapshot according to the profiles.
- **Implementation in Native Image.** We integrate our technique into the Native-Image building process (§3 and Appendix A). We develop a tracing profiler to produce object-ordering profiles

and discuss optimizations that reduce the build-time overhead and mitigate inaccuracies in matching objects.

- **Experimental evaluation.** We perform an evaluation of the new object-mapping technique, and the respective object reordering, on Native-Image binaries (§6). We measure page-fault reduction, execution-time speedup, and build-time overhead. We also analyze heap-ordering statistics to explain the effects of the proposed ordering strategy. As target workloads, we use the benchmarks from the AWFY suite [29] and seven popular microservice frameworks [12, 21, 30, 34, 37, 41, 48]. Our experimental results show that our improved object-mapping approach results in many fewer page faults (by 2.98× on average) and considerably improved startup time (by 1.98× on average) w.r.t. the original Native-Image implementation. Moreover, using the DaCapo [6] and Renaissance [40] benchmark suites, we show that our approach does not impair the performance of async/batch workloads where startup performance is not crucial and the startup time is very short w.r.t. steady-state execution time. Finally, we visually show the effects of our improved object-matching approach by presenting an analysis of the heap section of an optimized image (§7).

We start with background in §2, discuss correctness and efficiency in §5, and detail related work in §8. Finally, we discuss alternative approaches and limitations in §8, and conclude in §10.

2 Background

Here, we discuss Native-Image background and profile-guided reordering in a Native-Image binary.

2.1 Native Image

GraalVM Native Image [32, 50] (called *Native Image* for brevity) is a technology that uses AOT compilation to create a binary file from a JVM application, pre-initializing the Java environment at build time. Applications compiled via Native Image can be executed without instantiating a JVM. Native Image uses the Graal compiler [11] to produce the binary. Graal performs a pipeline of transformations on different *compilation units (CUs)*. Each CU is composed of a *root* (representing the method from which compilation started), and all the other methods that Graal inlined [39] into the root. The binary produced by Graal contains the compiled CUs (stored in the .text section of the binary) and a snapshot of pre-initialized Java heap memory (stored in the .svm_heap section of the binary). By default, Native Image orders objects in the heap snapshots as follows. First, CUs are ordered alphabetically, according to the signature of the root method. Then, Native Image detects, for each CU and following the previous order, all objects reachable from the CU, storing them in the .svm_heap section of the binary. This section is memory-mapped upon application execution—each page is lazily copied to memory upon first access.

Native Image supports *profile-guided optimizations (PGO)* to produce more efficient code. Native Image can create an *instrumented binary*, which contains code that collects metrics of interest (e.g., branch frequencies, virtual-call receiver types, and method-invocation counts), storing them in *profiles* upon application execution. Then, Native Image uses these profiles to produce an *optimized binary* (for the same application for which the instrumented binary was created).

Key Design Issues: Heap-Snapshot Divergences Between Builds. To generate the heap snapshot, Native Image locates the classes that are reachable from the classpath via a *points-to analysis* [19, 43, 50], and executes the static initializers of the classes that do not have observable side-effects. Then, Native Image traverses the object graph in a well-defined depth-first order, starting from the static fields of the reachable classes and the constants in the .text section. It is important to note that objects in the object graph and hence in the heap snapshot are very likely to differ across Native-Image compilations, even for the same application, for a number of reasons:

- **Non-determinism in running class initializers:** the execution of class initializers occurs in parallel and is subject to non-deterministic thread scheduling that potentially initializes classes differently across different runs.
- **Non-determinism in the Java Development Kit (JDK) and Java Class Library (JCL):** Some data structures (e.g. hash-tables, due to the default hash-code function being non-deterministic in JVM implementations) may be structurally different across compilations, even though their semantics remain the same.
- **Different compilation decisions:** Graal performs non-deterministic compilation decisions, in particular method-inlining decisions. Different inlining decisions can affect subsequent compiler optimizations, such as Partial Escape Analysis (PEA) [45] and constant folding.
- **Profiling influencing the contents of the binary:** If PGO is used, the instrumentation code inserted in the instrumented binary significantly alters the CUs (e.g., due to different inlining decisions) and the object allocations w.r.t. the optimized binary.

2.2 Profile-Guided Binary Reordering

The approach proposed in this article is related to prior work [4], which proposes to order code and objects in a Native-Image binary to decrease page faults and improve startup time. The above technique uses PGO, saving in the profiles the order in which the CUs are first executed and objects in the heap snapshot are first accessed. When generating a binary, the technique computes an ID for each object. Upon instrumented-binary execution, the technique collects the ID associated with each accessed object in the profiles. In the optimized build, the technique attempts to find, for each object in the heap snapshot, an object with the same ID from the profiles. If found, the two objects are considered semantically equivalent, and the information associated with the object in the profiles is used to position the corresponding object in the optimized binary.

From the above description, it follows that the efficiency in ordering objects in the heap snapshot (and hence in reducing page faults and startup time) depends on the ability to generate the same ID for semantically equivalent objects in different binaries. The technique proposes three different strategies to do so. The first assigns sequential IDs to objects, in the order in which they are encountered during heap snapshotting. The second identifies objects through hashes, computing them taking into account their type, their fields and their neighbours in the object graph. The third assigns an ID to an object based on its *heap path*, i.e., a list of all objects and arrays found on the path in the object graph that led to the inclusion of that object in the heap snapshot. The heap path is then hashed to obtain an ID for the object.

Problem. Unfortunately, none of the above strategies results in a satisfactory reduction of the application startup time. In the experiments reported by related work, the strategies can achieve only an average startup speedup of 1.07×, 1.09×, and 1.11×, respectively. These minor improvements stem from the poor capabilities of the strategies in generating the same ID for objects that are semantically equivalent in different binaries. The approach presented in this article tackles this problem, proposing a novel approach to generate the same ID for semantically equivalent objects that is more robust and efficient than the ones proposed by related work. Our technique is based on the HEAP PATH strategy and is described in the following section.

3 Design

In this section, we detail the design of the proposed heap-ordering technique.

Idea. The technique proposed in this work attempts to match two heap-object graphs by tracking paths that end with the nodes corresponding to the individual objects to be matched. Consequently, the technique matches objects by considering only (some of) their ancestors and not their siblings or children. The latter are considered only for objects of some specific types (detailed later) for

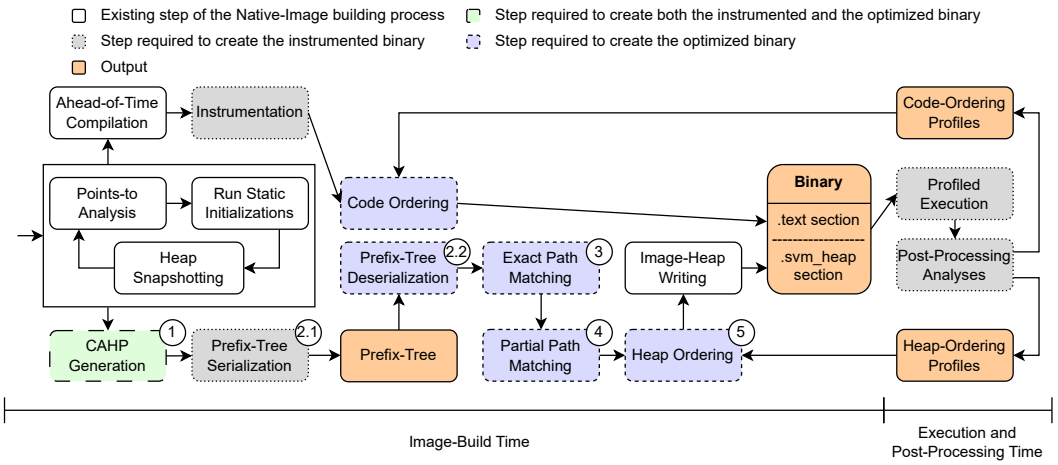


Fig. 2. Integration of our approach into the Native-Image build process.

which we compute an object value (§4.1). Simplifying the problem allows us to (1) design a simpler solution that does not need to consider the divergences between the entire neighborhoods of two nodes, which speeds up the matching time (and hence the build time), and (2) easily match the semantically equivalent objects associated with different paths across builds by computing their similarity (§4.4). The main challenge is in determining single paths in the two object graphs that accurately represent semantically equivalent objects.

Design Choices. We match each object in the optimized image with one object in the instrumented image. We do not consider the cases where (1) an object in the optimized image is not present in the instrumented image, and (2) an object in one image is split into multiple objects in the other image. To the best of our knowledge, no related work detects these cases, which are intrinsic in the heap-object graphs matching problem. In the former case, no information associated with the object to be optimized is collected during the instrumented execution, and hence no optimization may be performed. In the latter case, multiple objects would have different paths, and therefore one cannot find a match based on those paths, which is the goal of our approach. Since we are interested in ordering the objects in the optimized image, we do not map all objects in the instrumented image to objects in the optimized image.

Overview. Fig. 2 shows the integration of the proposed approach into the Native-Image build process. The number-annotated nodes represent the steps of our approach, performed either at instrumented- or optimized-image build time (except Step 1 that is executed at both). At instrumented-image build time, we produce metadata that is used at optimized-image build time. Below, we summarize the five steps of our approach:

(1) **Context-Augmented Heap-Path Generation** (§4.1): In this step, which takes place at both instrumented- and optimized-image build time, we create a list of elements that describes each object to be stored in the heap snapshot. This list represents a *context-augmented heap path* (CAHP), i.e., a path in the heap-object graph that is augmented with additional object-context information. CAHPs will be later used to compute object IDs and perform the matching between the objects in the instrumented and optimized images.

(2) **Prefix-tree Serialization/Deserialization** (§4.2): At instrumented-image build time, we create a prefix tree of the CAHPs (Step 2.1) and associate an ID with each CAHP (and hence with each object). This prefix tree is serialized and stored in a file. At optimized-image build time, we load this prefix tree from the file and deserialize it (Step 2.2).

(3) **Exact Path Matching** (§4.3): At optimized-image build time, we check whether some of the CAHPs of the objects in the optimized image correspond to some of the CAHPs of the objects in the instrumented image. To do so, we search for the CAHPs of the objects of the optimized image in the prefix tree. We call this search in the prefix tree *exact path matching* (henceforth also *exact matching* for short) since we match the objects of the optimized and instrumented images by checking the equality of their CAHPs. Objects associated with the matched CAHPs are associated with the IDs when building the instrumented image.
(4) **Partial Path Matching** (§4.4): Due to divergences between builds (§2.1), semantically equivalent objects may have different CAHPs in the instrumented and optimized images. In this case, since the CAHPs are not equal, the object is not exactly matched by the previous step. To mitigate the inaccuracies caused by these divergences, we exploit the prefix tree to perform a *partial path match* (henceforth also *partial match* for short): for each unmatched CAHP in the optimized image, we find the most similar unmatched CAHP in the prefix tree. We associate the object corresponding to the partially matched CAHP in the optimized image with the ID associated with the object corresponding to the partially matched CAHP in the instrumented image.
(5) **Heap Ordering** (§4.5): At optimized-image build time, we use the IDs computed by exact and partial matching to order the objects to be stored in the heap snapshot according to the profiles.

4 Approach

In each of the next sub-sections from §4.1 to §4.5, we detail one of the five steps of our approach.

4.1 Context-Augmented Heap-Path (CAHP) Generation

Here, we explain the context-augmented heap-path representation, starting with an example.

4.1.1 Example. Consider the Java application in Fig. 3. The main method (l. 6–15) checks whether the input arguments contain the command line option -v (l. 8 and 9). In this case, the application prints on the standard error the app version and terminates (l. 10–12). Otherwise, the execution continues with code omitted for brevity (l. 14). The major and minor versions of the app are retrieved by invoking methods left and right on the instance of type Pair (whose definition is reported in l. 1) stored in the static final field APP_VERSION (l. 5), respectively. The major and minor versions of the app are 1 and 0, respectively.

```
1  record Pair<L, R>(L left, R right) {}
2
3  public class Main {
4   private static final Pair<Integer, Integer>
5       APP_VERSION = new Pair<>(1, 0);
6   public static void main(final String[] args)
        {
7    final List<String> argsList =
8       Arrays.asList(args);
9    if (argsList.contains("-v")) {
10     System.err.println("v" +
           APP_VERSION.left() +
11         "." + APP_VERSION.right());
12     System.exit(0);
13   }
14   // other code omitted for brevity...
15  }
16 }
```

Fig. 3. Example Java application that prints the application version.

Fig. 4 reports the subgraph of the heap created by the expression new Pair<>(1, 0) (l. 5) in the example application. The Pair instance (identified in this example by the label 1 and hence named Pair@1 for clarity) stores the Integer 1 (label 2) in the field left and the Integer 0 (label 3) in the field right. Since the Pair reference is stored in a static final field, consider the case where this reference is constant-folded into method Main.main(String[]). Fig. 5 shows the CAHP for Integer@2. Each node represents an element of the CAHP and reports inside the associated value (as later detailed in Table 1). The index of the element in the list is reported on the left side between square brackets, before the element type. The CAHP associated with Pair@1 simply consists of the first two nodes

Table 1. Element types composing a context-augmented heap path (CAHP).

Element Type Name	Value	Previous Element Type	Next Element Type	Description
Method	Method signature including declaring class	-	Object	Method embedding a constant object reference.
Static Field	Field signature including holder class	-	Object	Static field storing an object reference.
Data Section	-	-	Object	The next object is stored in the data section of the binary.
Resource	-	-	Object	The next object represents a resource.
InternString	-	-	Object	Indicates that the next object is an interned string.
Object	Fully qualified name of the object class	Method, Static Field, Data Section, Resource, Instance Field, or Array Index	Instance Field, Array Index, Object Value, or ε	Object (or array) included in the heap snapshot.
Instance Field	Field signature including holder class	Object or Object Value	Object	Indicates that the preceding object stores the next object in the given field.
Array Index	Index value	Object	Object	Indicates that the preceding object (an array) stores the next object at the given index.
Object Value	Value representing the preceding object	Object	Instance Field or ε	A value (string) representing the preceding object.

shown in Fig. 5, while the CAHP associated with Integer@3 contains field Pair.right at index 2 and value 0 at index 4.

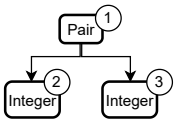


Fig. 4. Example heap graph for the expression new Pair<>(1, 0).

4.1.2 Description. We associate each object with a list of elements computed from the first path in the heap-object graph to that object found by Native Image when performing the depth-first traversal to determine the reachable objects to include in the heap snapshot (§2.1). The list ends with that object. Later, we use these lists to associate objects with IDs. This allows us to (1) reduce collisions (i.e., the association of the same ID with semantically different objects), (2) produce debugging and visualization metadata (§7), and (3) perform partial matching by checking for similar objects (§4.4). In addition to the information encoded in the path of the heap-object graph, our strategy includes the value encoded by objects of certain types (explained below). For this reason, we say that each object is associated with a *context-augmented heap path* (CAHP) represented by the list. We design CAHPs to minimize collisions and maximize similarity. To minimize collisions, we map as few objects as possible to the same CAHP, while to maximize similarity, we map semantically equivalent objects to the same CAHP or the most similar CAHPs across builds.

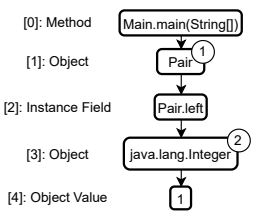


Fig. 5. Context-augmented heap path (CAHP) associated with Integer with label 2 in Fig. 4.

CAHP Structure. Table 1 shows all the element types composing CAHPs. Each element type is identified by a name, has a description, potentially stores a value that represents the element, and potentially defines a relationship with the previous element, the next element, or both. We report the relationships as a grammar for CAHPs in Appendix D. We use ε to denote the end of the list and hence no element. A CAHP starts with a root element type indicating the reason why the first object is a root in the heap object graph, namely Method, Static Field, Data Section, Resource, and InternString. Native Image traverses the object graph in a well-defined depth-first order, starting from the objects stored in required static fields of the reachable classes (Static Field), the data-section constants (Data Section), the constants in the code section (Method), the interned strings (InternString), and the resources (Resource). For example, the first element can be a method in which the object reference was inlined, and its value is the method signature. The second element of the list must be the object whose reference was

inlined (element type Object) represented by the fully qualified name of its type. Depending on the object and heap traversal, the next element can be:

(1) An Instance Field, if the object is a Java object (and not an array) that stores (in a field) a reference to another object and the heap-traversal algorithm traverses this reference. Instance fields allow distinguishing multiple instances of the same type, referenced by the same object.
(2) An Array Index, if the object is an array that stores (at a certain index) a reference to another object and the heap traversal algorithm traverses this reference. Array indexes serve a similar purpose to instance fields, i.e. clearly distinguishing instances of the same type, referenced by the same array instance.
(3) An Object Value, if the object is an instance of a known class, a class that encodes metadata (e.g., java.lang.Class or java.reflect.Method), a boxed primitive, an enum, or a string. The object value is an optional element that must be followed by either Instance Field or ε.
(4) ε, if the last object is the final object in the CAHP.

Elements of these types and Object (except ε) repeat until the end of the list. We found that it was important to introduce the Object Value element to CAHPs, because this allows distinguishing objects more precisely (e.g. singleton objects that represent internal metadata), and it improves matching. Moreover, CAHPs computed for PGO builds do not rely on Array Index elements, which are used only for debugging purposes. The reason is that array indices are too volatile across Native-Image builds, and using them would degrade PGO improvements.

Collisions. Without employing array indices, the same CAHP may be associated with multiple objects, leading to collisions as discussed in §6.5 and §7. In this context, Object Value elements are also useful for matching boxed primitives stored at different array positions. §6 shows the performance impact of using Object Value elements.

Algorithm 1: Context-Augmented Heap-Path Function

```
Function contextAugmentedHeapPath(object, og):
computes the context-augmented heap path for the
provided object
Input:
object, the object for which the context-augmented heap
path has to be computed
og, the heap object graph containing the provided object
Output:
the context-augmented heap path for the provided object
1  cahp ← new CAHP()
2  current ← object
3  while true do
4      if shouldCreateObjectValue(current) then
5          cahp.prepend(new ObjectValueElement(current))
6      cahp.prepend(new ObjectElement(current))
7      if og.isRoot(current) then
8          rootElement ← newRootElement(current)
9          cahp.prepend(rootElement)
10         break
11     else
12         parent ← og.parents(current).first()
13         if current instanceof Array then
14             index ←
                 og.accessedArrayIndex(parent, current)
15             cahp.prepend(new ArrayIndexElement(index))
16         else
17             field ←
                 og.accessedField(parent, current)
18             cahp.prepend(new InstanceFieldElement(field))
19         current ← parent
20 return cahp
```

4.1.3 Algorithm. The pseudocode to create CAHPs is shown in Algorithm 1. The algorithm takes as input an object (either an Object or an array reference) and the heap-object graph that contains the object. The graph includes reverse links for each object, i.e., a list of all objects pointing to it. The algorithm is bottom-up: it traverses the object graph from *object* up to one root and prepends the elements to the CAHP, allowing implementations that exploit dynamic programming (Appendix A). The first parent returned by an object is always the root or an object that leads to a root (i.e., in our algorithm we cannot encounter cycles).

The algorithm first allocates the empty CAHP (l. 1) and then iteratively traverses the first path in the heap object graph from the object to the root (l. 2–19). For each object on the path (l. 2), we check whether we should create an Object Value via the helper function shouldCreateObjectValue (l. 4) that inspects the type of the object (§4.1.2). If the check succeeds, we create an Object Value and prepend the element to the CAHP. Then, we prepend an Object element for the current object. If the object is a root, we prepend the first element (of type Method, Static Field, Data Section, Resource, or InternString) by leveraging the helper function newRootElement (omitted for brevity) and break the loop (l. 7–10). If the object is not a root, we obtain its parent in the path (l. 12). If the parent is an array (l. 13), we obtain the array index where the current object is stored (l. 14), and we prepend it to the CAHP (l. 15). Otherwise, the parent is an object instance. Hence, we obtain the field where the current object is stored (l. 17), and we prepend the field to the CAHP (l. 18). We then repeat this for the parent (l. 19). Finally, we return the computed CAHP (l. 20).

4.2 Prefix-Tree Serialization/Deserialization

After computing the set of CAHPs, we construct the prefix tree that will be later employed to facilitate matching.

4.2.1 Example. Fig. 6 shows the prefix tree (also known as trie) produced by the CAHPs of the heap graph from Fig. 4 (§4.1.1). The prefix tree encodes the three CAHPs associated with the three objects (Pair@1, Integer@2, and Integer@3). The last node of each CAHP is associated with a CAHP-unique numeric ID that identifies the corresponding object. IDs indicate that the prefix tree has been built by inserting first Pair@1, then Integer@2, and finally Integer@3.

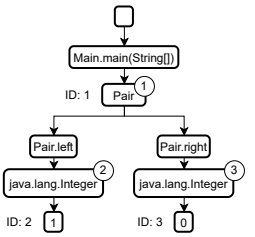


Fig. 6. Prefix tree produced by the context-augmented heap paths (CAHPs) of the heap graph in Fig. 4.

4.2.2 Description. In the instrumentation build, we use the CAHPs to build a prefix tree where each node corresponds to a CAHP element. A prefix tree allows fast lookup of CAHPs, enabling efficient matching of optimized-image CAHPs with instrumented-image CAHPs. We iterate over all the CAHPs and insert each of them into the prefix tree. Upon insertion, we associate the node in the prefix tree that corresponds to the last element of the CAHP with an ID. As a consequence, not only leaf nodes but every node of the prefix tree whose element is of type Object or Object Value can be associated with an ID (a CAHP can be a prefix of another CAHP). The same ID is stored in the object associated with that CAHP so that it can be profiled at runtime by instrumentation code. To generate the IDs, we keep a counter (starting from 1), incremented for each inserted CAHP. Value 0 is reserved and used as null. In the instrumented build, the prefix tree is serialized and dumped to a file. In the optimized build, the prefix tree is loaded from the file and deserialized. Since the generation of a prefix tree is standard, we report the pseudocode in Appendix B.

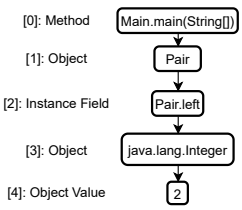


Fig. 7. Context-augmented heap path (CAHP) associated with Integer encapsulating value 2 for expression new Pair<>(2, 0).

4.3 Exact Path Matching

In this section, we show an example of exact path matching and then the description of this step.

4.3.1 Example. Consider a scenario in which the application version has been updated from $(1, 0)$ to $(2, 0)$. The optimized-image build constant-folds an object reference that corresponds to new Pair<>(2, 0) into the method Main.main(String[]) instead of new Pair<>(1, 0) (Fig. 3, l. 5). Assume now that the optimized-image build uses profiles (and the prefix tree) that were obtained from a

previous version of the application (e.g. profiles could be collected periodically on a production deployment). When generating CAHPs for objects in the heap snapshot, the optimized-image build generates exactly the same CAHPs from the prefix tree in Fig. 6 for the Pair object and the Integer object from the right field (label 3, encapsulating int 0). Since these CAHPs are in the prefix tree, exact matching finds the corresponding prefix-tree nodes and associates these objects with IDs 1 and 3, respectively. However, the CAHP associated with the Integer stored in the left field (Fig. 7) is different from the CAHP shown in Fig. 5— the last element of the newly generated CAHP stores 2 instead of 1. Since this CAHP is not in the prefix tree, exact matching indicates a failure by returning the ID 0. The CAHP remains unmatched and the corresponding object lacks an ID. We handle this unmatched CAHP in §4.4.1.

4.3.2 Description. In the optimized build, we match the objects in the snapshot of the optimized image with the objects in the snapshot of the instrumented image (and hence in the profiles collected by the instrumented execution of the application). Matched objects (and hence CAHPs) are assigned the same ID. To do so, we perform *exact path matching*, which pairs the objects in the optimized image with objects in the instrumented image by checking for the equality of their CAHPs. To be equal, two CAHPs must consist of the same elements. The CAHPs of the objects of the instrumented image are stored in the prefix tree, while the CAHPs of the objects of the optimized image are kept by Native Image. We iterate over all the CAHPs of the optimized image and we check whether each of these CAHPs is in the prefix tree. The CAHPs that are found in the prefix tree are assigned the ID associated with the last element of the CAHP in the tree. Since exact path matching represents a lookup in the prefix tree, a well-known operation, we report the pseudocode of exact path matching in Appendix C.

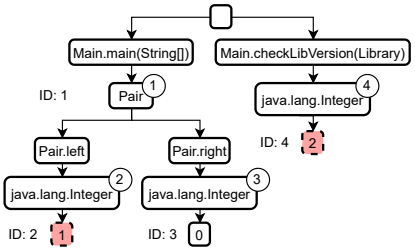


Fig. 8. Extension of the prefix tree in Fig. 6 for a more comprehensive example of partial matching.

4.4 Partial Path Matching

In this section, we report an example, the description, and the algorithm of partial matching.

4.4.1 Example. We perform a partial match for the unmatched CAHP discussed in §4.3.1 and reported in Fig. 7. To present a more comprehensive example, we consider the prefix tree in Fig. 8, which extends the one shown in Fig. 6 by adding a subtree for Integer with value 2 (whose ID is 4), which is constant-folded into another method Main.checkLibVersion(Library). The nodes corresponding to the unmatched CAHPs stored in the prefix tree are reported in pink with dashed borders. Partial matching traverses the prefix tree starting from these nodes, that is, the nodes whose object type corresponds to the object type of the provided CAHP. Node associated with ID 3 is ignored because the corresponding CAHP has already been exactly matched. While traversing the prefix tree, elements must match, except for Object Value and Array Index elements. To match the CAHPs that most likely correspond, partial matching relies on a score function (explained in §4.4.2). In particular:

- Starting from the node associated with ID 2 (which stores value 1), partial matching computes the suffix of the CAHP consisting of nodes [Main.main(String[]), Pair, Pair.left, java.lang.Integer, 1] of length 5. To compute this suffix, partial matching uses all elements of the provided CAHP. Among these nodes, only 4 of them have been matched upon traversal. The last node of the suffix has not been matched due to a different Object Value element. Partial matching associates this suffix with a score of 4, i.e., the number of matched nodes.

- Starting from the node associated with ID 4 and storing value 2, partial matching computes the suffix consisting of nodes [java.lang.Integer, 2] of length 2. To compute this suffix, partial matching uses only the last two elements of the provided CAHP. This is because the third-to-last element of this CAHP (Pair.left of type Instance Field) does not match the element of the parent of the node corresponding to Integer with label 4 (Main.checkLibVersion(Library) of type Method) and the elements are neither an Object Value nor an Array Index. Since all the nodes of the suffix have been matched upon traversal, partial matching associates this suffix with a score of 2.

Given the scores of these suffixes, partial matching determines that the CAHP in the optimized image most likely corresponds to the CAHP associated with ID 2 in the instrumented image, i.e., the CAHP whose suffix has the highest score (score 4 vs. 2). The object associated with the first CAHP is associated with ID 2.

Algorithm 2: Partial Path Match Function

```
Function partialPathMatch(root, cahp):
finds the unmatched CAHP in the prefix tree that most
likely corresponds to the given CAHP by performing partial
path matching
Input:
root, the root of the prefix tree
cahp, the CAHP to search for in the prefix tree
Output:
the ID associated with the unmatched CAHP in the prefix
tree that most likely corresponds to the given CAHP or 0 if
no suffix is found
1  maxScore ← 0
2  maxId ← 0
3  reversedCahp ← reverse(cahp)
4  startNodes ← [
5   node for node in nodes(root)
6   if node.isUnmatched() and sameObjectType(node, cahp)
7  ]
8  foreach start in startNodes do
9      current ← start
10     score ← 0
11     foreach element in reversedCahp do
12         match ← current.matches(element)
13         if match then
14             score++
15         else if not current.isObjectValueElement()
16          or not element.isObjectValue() then
17             break
18         current ← current.getParent()
19     if score > maxScore then
20         maxScore ← score
21         maxId ← start.getId()
22 return maxId
```

4.4.2 Description. As explained in §2.1, due to divergences between the instrumented and optimized builds, the optimized image may contain objects that are not present in the instrumented build and vice versa— there is no one-to-one mapping between objects of two images and an object of one image corresponds at most to an object of the other image (§3). Moreover, the semantically equivalent object in the optimized and instrumented images may be connected differently in the object graph or even store different values (§2.1). These objects may have different CAHPs between images and may hence not be matched exactly. The goal of the *partial path matching* step is to match these objects, mitigating the inaccuracies caused by divergences between builds. To do so, we find for each unmatched CAHP in the optimized image the unmatched CAHP in the instrumented image that most likely corresponds to the first. We evaluate the similarity between CAHPs using the prefix tree and we associate the object corresponding to the partially matched CAHP in the optimized image with the ID associated with the object corresponding to the partially matched CAHP in the instrumented image. Partial matching follows the heuristics and rules listed below:

(1) When performing partial matching, we exclude all CAHPs that have been matched exactly in the previous step. This is because we assume that exact path matching always finds "correct" matches— we assume that most of the semantically different objects are not associated with the same CAHP. Excluding exactly matched objects also allows us to reduce the number of comparisons, speeding up the algorithm and the image build.

(2) Since partial matching provides results based on similarity, partial matching may map multiple objects with different CAHPs in the optimized image to the same object in the instrumented image, assigning the same ID.
(3) Similarity is based on three rules:
   (a) The two unmatched CAHPs of the semantically equivalent object in two images likely have a common suffix but not a common prefix. This is because an ancestor of this object may have been found differently, e.g., because an object has not been constant-folded into a method but reached through another object. This may happen due to differences in the inlining decisions, which affect Partial Escape Analysis [45]. Another source of differences in CAHPs is that the reachability analysis may decide to prune different object fields [51]. We use the first path to an object that Native Image discovers, and this path may vary between instrumented and optimized builds.
   (b) The semantically equivalent object may acquire different values in two images. This can be caused by parallel execution of static class initializers during build-time, by build-time initialization of classes that depend on the system state, or by hash-code nondeterminism. When computing similarity between CAHPs, we consider CAHPs with equal Object Value elements to be similar, but we do not exclude CAHPs with different Object Value elements.
   (c) Divergences between builds are unlikely to store a semantically equivalent object in different fields. Hence, we expect Instance Field elements to correspond in common suffixes.

Given an unmatched CAHP in the optimized image, partial matching traverses the prefix tree bottom-up, starting from all the nodes that correspond to unmatched CAHP (Rule 1) in the instrumented image and whose object type corresponds to the object type associated with the CAHP in the optimized image. For each node, partial matching finds the longest *partial suffix* of the two CAHPs (henceforth called suffix for brevity, according to Rule 3a). In this suffix, all the elements of the two CAHPs match (Rule 3c) except for elements of type Object Value that may not match (Rule 3b). The starting node may be of element type Object but also Object Value. Then, partial matching assigns a score to each suffix, computed as the number of elements in the suffix that match between the two CAHPs. In practice, a suffix can have a score of at least *length(suffix) - numberOfObjectValueElements(suffix)* and at most *length(suffix)*. Partial matching associates the unmatched CAHP in the optimized image with the ID of the CAHP corresponding to the suffix with the highest score. We note that partial matching may incorrectly identify objects as similar. In the case of heap ordering, incorrectly matched objects do not compromise application correctness but potentially lead to incorrectly ordered objects (§5). In §6, we show that partial matching effectively reduces page faults and improves performance.

4.4.3 Algorithm. The pseudocode for partial matching is shown in Algorithm 2. The algorithm takes the root of the prefix tree (from the instrumented build) and the CAHP of the object to match (from the optimized build) as inputs, and returns as output the ID associated with the given CAHP or 0 if no suffix of the given CAHP is found. The algorithm first defines variables *maxScore* and *maxId* (l. 1 and 2). These variables are used to keep track of the highest score among the suffixes and the corresponding ID. Since we traverse the prefix tree bottom-up, the algorithm reverses the CAHP provided as input (l. 3). Then, the algorithm collects the nodes in the prefix tree that are associated with the unmatched CAHPs whose corresponding object is of the same type as the object represented by the CAHP to match (l. 4–7). These nodes are used to start the traversals (l. 8–9). In our implementation, we maintain a map to efficiently find these nodes without traversing the entire prefix tree. For each traversal, we define a *score* variable that stores the score associated with the longest suffix (l. 10). The traversal is performed by iterating over the reversed elements of the given CAHP (l. 11). For each element, the algorithm checks whether the element of the current node

matches the current element (l. 12). If the elements match (l. 13), the algorithm increases the score by 1 (l. 14). If the elements do not match and at least one of the two is not of type Object Value (l. 15 and 16), the algorithm has found the longest suffix and hence breaks the loop (l. 17). Otherwise, the algorithm proceeds with the traversal of the suffix by obtaining the parent of the current node (l. 18). After the traversal, the algorithm checks whether the score associated with this suffix is greater than *maxScore* (l. 19). If it is, the current score replaces the *maxScore* (l. 20) and the ID associated with the last node of this suffix becomes the potential ID to be returned (l. 21). If several suffixes are associated with the highest score, only the first suffix is considered. After processing all the suffixes, the algorithm returns *maxId* (l. 22). The complexity of the partial matching algorithm is $O(n \cdot m)$ where n is the number of unmatched CAHPs in the prefix tree and m is the average longest-suffix length. To reduce the execution time of the algorithm, our implementation exploits several optimizations, which we briefly discuss in Appendix A.

4.5 Heap Ordering

4.5.1 Description. Given a list of objects and *access profiles (APs)*, consisting of a map between the object ID and the access index, the objective of this step is to order the list by computing an ascending *ordering index (OI)* for each object. OIs do not need to be contiguous—gaps in OIs do not introduce gaps between objects in the produced binary. Moreover, multiple objects may be assigned to the same OI, in which case they are placed one after the other in the binary. We conceptually order the objects into four groups, i.e., we place objects with similar properties close together, with the goal of further reducing page faults. To one OI corresponds a single group. Only within the first two groups, objects are placed according to their access index. The object-ordering groups are defined as follows:

(1) The first group contains exactly-matched objects whose IDs are in the APs. Within this group, objects are ordered according to the index specified in the APs. Hence, the ordering indices of this group range from 0 to APs.size(). We place these objects close together since we assume that exact path matching finds "correct" matches—we expect these objects to cause page faults.
(2) The second group contains partially-matched objects whose IDs are in the APs. Within this group, objects are ordered according to the index specified in the APs. The ordering indices of this group range from APs.size() to APs.size() * 2. We place partially- and exactly-matched objects in different groups because partial path matching may be subject to matching errors by design. We expect some of the objects in this group to not cause page faults.
(3) The third group contains objects whose IDs are in the APs but whose sizes exceed the page size. All objects in this group have OI equal to APs.size() * 2 + 1. This group limits the separation of small objects stored in groups 1 and 2 across multiple pages, potentially reducing page faults.
(4) The fourth group contains objects whose IDs are not in the APs. Within this group, objects maintain the default ordering provided by Native Image. We specify −1 as the OI for the objects in this group.

4.5.2 Algorithm. The pseudocode of our orderingIndex function is shown in Algorithm 3. Given the heap-ordering profiles, the orderingIndex function returns the OI associated with the *object* provided as a parameter. First, the algorithm obtains the ID associated with the provided object (l. 1), and checks whether the ID is in the profiles (l. 2). In this case, the algorithm checks whether the object size exceeds the page size of the OS for which the binary must be generated (l. 3). If it does, the algorithm computes and returns the OI associated with the third group (l. 4). If the ID is in the profiles and the object size does not exceed the page size, the algorithm checks whether the ID has been obtained via a partial match (l. 5) by calling the isPartiallyMatched helper (omitted for brevity). In that case, the algorithm returns the sum of the OI reported in the profiles and the size

of the profiles, placing the object in the second group (l. 6). If the ID is in the profiles, the object size does not exceed the page size, and the ID is derived via an exact match, the algorithm returns the index reported in the profiles placing the object in the first group (l. 8). Finally, if the ID is not in the profiles, we place the object in the fourth group by returning −1 (l. 9).

4.5.3 Example. Consider the exact and partial matching performed in §4.3.1 and §4.4.1, respectively, and the following APs:

```
1: 1, // [ID: 1, accessIndex: 1] Pair
2: 2, // [ID: 2, accessIndex: 2] Pair.left
3: 3  // [ID: 3, accessIndex: 3] Pair.right
```

Access indices in the APs indicate that, in the execution of the instrumented binary, the Pair object (with ID 1) has been accessed first, followed by the Integer stored in the left field (with ID 2) and the Integer stored in the right field (with ID 3). The Integer with ID 4, reported in Fig. 8 has not been accessed at runtime and hence is not in the APs. Our orderingIndex function associates object 1 with OI 1 (exact match), object 3 with OI 3 (exact match), and object 2 with OI 5 (= 3 + 2 computed as $APs.size() + APs.get(2)$) due to partial matching. The object associated with ID 4 is not in the APs and hence is associated with OI 7 (= 3∗2+1 computed as $APs.size() * 2 + 1$). We produce a binary that stores the objects with IDs 1–4 in this order: [1, 3, 2, 4].

Algorithm 3: Ordering Index Function

```
Function orderingIndex(aps, object):
computes the ordering index for the provided object
Input:
aps, a map between profiled IDs and access indices
object, the object to be ordered
Output:
a number representing the ordering index for object or −1 if the object should not be ordered
1 objectId ← id(object)
2 if aps.contains(objectId) then
3     if size(object) >= OS.getPageSize() then
4         return aps.size() * 2 + 1
5     else if isPartiallyMatched(objectId) then
6         return aps.size() + aps.get(objectId)
7     else
8         return aps.get(objectId)
9 return −1
```

5 Correctness and Efficiency

Correctness. As shown in Figure 2, our approach performs the heap ordering before image-heap writing. The heap-ordering step does not modify the heap object graph but its serialized representation, i.e., we do not modify references between objects but define the order in which these objects will be stored in the binary. Native Image allows storing the heap object graph in one of the $N!$ different serialized representations, where N is the number of objects in the heap object graph. All the serialized representations are correct. We choose a representation that is optimized for reducing page faults. Consequently, no code location or object points to the wrong objects, preserving program semantics. The image-heap writing step writes the given heap object graph into the binary, preserving the given object order and references. Algorithm 4 shows the pseudocode of the image-heap writing step. The algorithm takes as input a binary writer, used to dump the output native-image binary, and the heap-object graph containing all objects to be stored in the binary, and returns as output the mapping from the objects in the heap object graph to the offsets in the binary. The algorithm starts by obtaining the ordered set of objects to write to the output binary (l. 1). In instrumented and regular builds, objects are returned in the default order (§2.1), while in optimized builds, objects are returned sorted by ordering index (§4.5). Then, the algorithm initializes a map between objects and binary offsets (l. 2), writes each object to the output binary, and obtains the offset in the binary where each object has been stored (l. 3–4). In this step, objects are written to the output binary by replacing references stored in fields and array locations with blanks. The reason is that the referenced objects may not have already been written to the binary and hence cannot be pointed to. Written objects are put in the map and associated with the

returned offset (l. 3). After writing all objects in the heap object graph, the algorithm loops over all the objects again (l. 6). For each referenced field, the algorithm obtains its offset (l. 7) and its object references in the heap object graph, i.e., the pointed objects (l. 8). For each object reference (l. 9), the algorithm obtains the associated offset (l. 10) and invokes a fillReferences method (l. 10), whose purpose is filling the references to the object (left blank on purpose) in the fields and array locations of the current object. Filling references after dumping the objects allows maintaining the same heap object graph independently of the object order. Finally, the algorithm returns the mapping from object to offset (l. 11–12). Native Image will use this mapping to fill references in the code to objects in the heap snapshot.

Algorithm 4: Write Image Heap Function

```
Function writeImageHeap(bw, og):
writes the given object graph to the output binary
Input:
bw, helper binary writer used to produce the native-image binary
og, the heap object graph to be written
Output:
returns the mapping from the objects in the heap object
graph to the offsets in the binary and modifies the output native-image binary via bw
1  objects ← og.getObjects()
2  obj2Offset ← new IdentityMap()
3  foreach object in objects do
4      offset ← bw.write(object)
5      obj2Offset.put(object, offset)
6  foreach object in objects do
7      objOffset ← obj2Offset.get(object)
8      children ← og.getChildren(object)
9      foreach child in children do
10         childOffset ← obj2Offset.get(child)
11         bw.fillReferences(object, objOffset,
12                           child, childOffset)
13 return obj2Offset
```

Efficiency. The proposed approach produces correct binaries, but their object layout is potentially not optimal. Ordering of multiple objects with the same ID (§6.5) and potential matching inaccuracies, i.e., wrong matches between objects in the instrumented and optimized images, may lead to suboptimal ordering and hence slower startup. In the case of optimal ordering, all and only the runtime-accessed objects are placed in a few contiguous pages. In the case of suboptimal ordering, runtime-accessed objects may be interleaved with objects that are not accessed, leading the OS to fetch more pages than what would be needed in the case of optimal ordering.

6 Evaluation

In this section, §6.1 presents the evaluation settings. Then, §6.2 analyzes the page-fault-reduction, §6.3 presents the execution-time speedup, and §6.4 shows the build-time overhead introduced by the proposed technique. Finally, §6.5 presents heap-ordering statistics that provide insights into how effective object-matching and heap ordering are.

6.1 Evaluation Settings

We run our experiments on a machine equipped with a 16-core Intel Xeon Gold 6326 (2.90 GHz), 256 GB of RAM (8x32GB, 3200MHz), and Solid-state Drive (SSD, NVMe, U.2, sequential read/write 6800/2700 MB/s, and random read/write 850K/130K IOPS), 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 the GraalVM Community Edition, based on OpenJDK 21, using the Graal compiler. We modify both Graal and Native Image to implement our technique. We collect measurements in an isolated environment with minimal perturbation, where (almost) no other process is being executed.

The FaaS model assumes that user workloads consist of often short-running programs that are invoked to handle individual requests. We use the AWFY [29] suite to evaluate the startup-performance impact, because it consists mainly of short-running programs. The 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 several widely-used microservice frameworks, particularly *Helidon* [34], *Ktor* [21], *Micronaut* [30],

Play Framework [37], *Quarkus* [41], *Spring* [48], and *Vert.x* [12]. The computation required to compute the response is minimal, and most of the time is spent on starting the Native-Image binary and initializing the framework. We use *helloworld* because we want to measure startup improvements in the microservice frameworks, and not in the user application, which we already evaluate using AWFY. We also report results on a *helloworld* implemented in Java that uses the HttpServer class provided by the Java Class Library. We call this workload *vanilla*. We build statically linked executables [26]. To evaluate the first binary execution in which data is not already present in RAM and needs to be fetched, we drop OS caches, as well as reclaimable slab objects such as *dentries* and *inodes* between workload iterations [47]. We employ a page size of 4 KB.

To better analyze the impact of the proposed approach to map semantically equivalent objects, in addition to the unmodified baseline, we compare with related work [4] and we evaluate 5 different ordering strategies:

(1) *Code* (CODE for short): This strategy corresponds to the CU code-ordering strategy proposed in related work and is useful to compare the impact of code-ordering and heap-ordering alone.
(2) *Exact+Object Values+Partial* (HEAP for short): In this strategy, we order the heap using exact and partial matching on CAHPs that contain object values. This strategy is useful to understand the impact of heap-ordering alone.
(3) *Code+Exact* (EXACT for short): In this strategy, we order both code and heap. The heap is ordered employing only exact matching, and CAHPs do not contain object values.
(4) *Code+Exact+Object Values* (OBJECT VALUES for short): This strategy is similar to EXACT except for CAHPs that contain object values. This strategy is useful to validate that including object values increases performance.
(5) *Code+Exact+Object Values+Partial* (CODE+HEAP for short): This strategy has all the features presented in this article and the code ordering proposed by related work. This is the strategy that one would like to release in production.

For each strategy (including the unmodified baseline) and workload, we build 3 images. For each image, we run 33 iterations to measure page faults and 33 iterations to measure the end-to-end execution time (for AWFY, using perf) and the elapsed time until the first response (for microservices). To report the elapsed time until the first response, we 1) obtain the start timestamp, 2) run the microservice workload in parallel, 3) continuously ping the microservice endpoint using curl until receiving a response, 4) obtain the end timestamp, and 5) kill the microservice workload. The elapsed time is computed by subtracting the end and start timestamps. An iteration is a single execution of the workload in a separate process. To determine page-fault reductions, we first trace page faults by executing the command "perf trace -F all – <benchmark_command>" and then we process the trace to count the page faults caused by the Native-Image binary, in particular to the .text and .svm_heap sections. In both cases, we compute the average of all the measurements.

To show that the proposed technique does not impair performance of async/batch workloads where startup performance is not crucial and the startup time is very short w.r.t. steady-state execution time, we evaluate page-fault reductions and (end-to-end) execution-time speedups of strategy CODE+HEAP on the DaCapo [6] and Renaissance [40] benchmark suites. In particular, we run all the DaCapo and Renaissance benchmarks that Native Image supports in our evaluation settings—Native Image does not support Apache Spark benchmarks [33, 35, 55]. We build 3 native images, and for each image, we run 10 iterations to measure page faults and 10 iterations to measure the end-to-end execution time. We note that optimizing DaCapo and Renaissance benchmarks that run for several seconds is not the aim of this article, and we expect no slowdown nor speedup—Serverless and FaaS workloads are typically short REST APIs that run for a few dozen milliseconds [10, 15, 46].

All figures (except those in §6.4, Appendix E, and Appendix F) report factors computed as $M_{baseline}/M_{optimized}$, where $M_{baseline}$ refers to the average measurement obtained on the unmodified

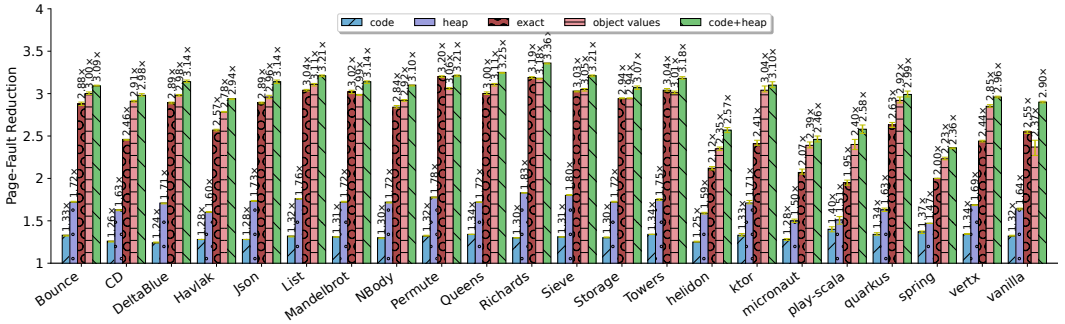


Fig. 9. Page-fault reduction achieved by the ordering strategies.

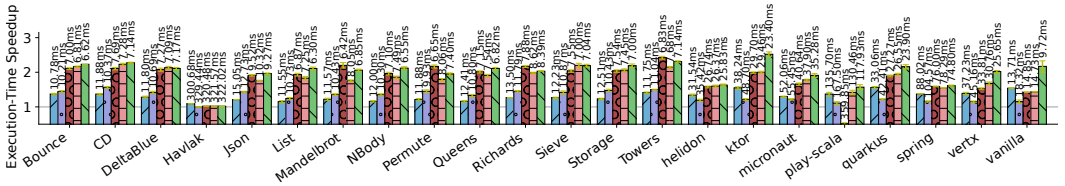


Fig. 10. Execution-time speedup achieved by the ordering strategies.

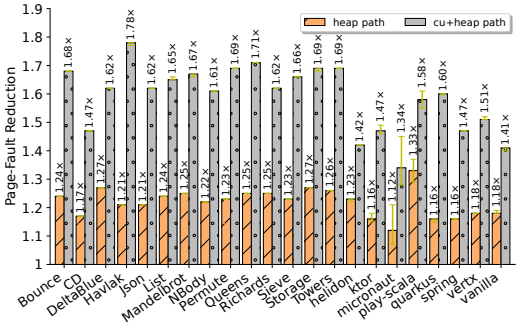


Fig. 11. Page-fault reduction achieved by related work.

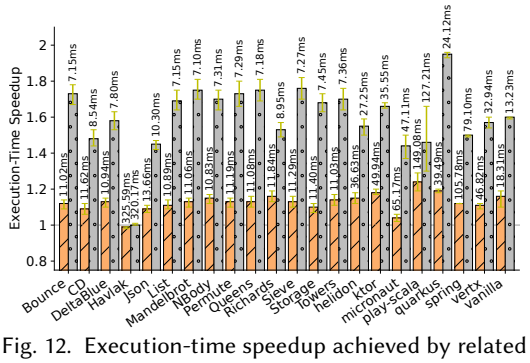


Fig. 12. Execution-time speedup achieved by related work.

Native Image and $M_{optimized}$ refers to the one obtained using one of our evaluated techniques (higher is better). First, we report the AWFY benchmarks (capitalized), then the workloads that employ microservices (lowercase), and finally *vanilla*. Above each bar, we report the exact factor, except for figures reporting execution-time speedups where we report the absolute execution time in milliseconds. The yellow error bars represent 95% confidence intervals (CI) of the measurements.

6.2 Page-Fault Reduction

Fig. 9 shows the page-fault-reduction factors for the evaluated ordering strategies. Experimental results show that HEAP reduces more page faults than CODE with average reduction factors of 1.67× and 1.31×, respectively. This confirms the importance of ordering heap snapshots and matching objects. Page-fault reduction for CODE ranges from 1.24× (*DeltaBlue*) to 1.40× (*play-scala*), while factors for HEAP range from 1.46× (*spring*) to 1.83× (*Richards*). While reduction factors associated with code ordering are similar on AWFY and microservices, factors associated with heap ordering are slightly lower on microservices than AWFY. Strategies EXACT, OBJECT VALUES, and CODE+HEAP show that code and heap ordering are synergistic, with average reduction factors of 2.65×, 2.82×, and 2.98×. The reason is that, in Native Image, metadata stored in the heap is affected by code ordering. Similar to the HEAP strategy, combined code- and heap-ordering reduction factors are

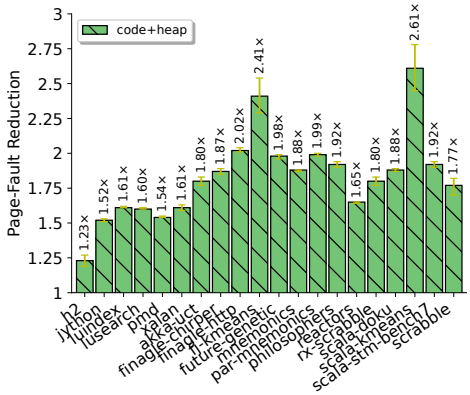


Fig. 13. Page-fault reduction achieved by strategy CODE+HEAP on DaCapo and Renaissance.

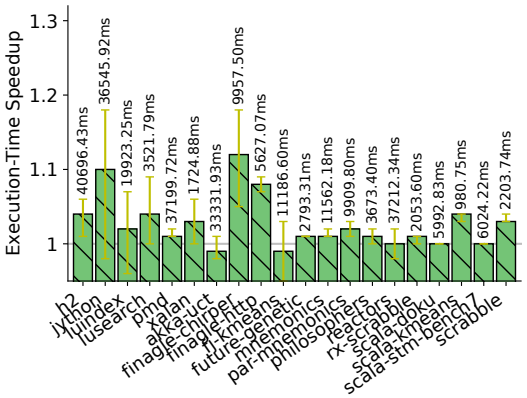


Fig. 14. Execution-time speedup achieved by strategy CODE+HEAP on DaCapo and Renaissance.

slightly lower on microservices than AWFY. Results for OBJECT VALUES show that Object Values elements of CAHPs are effective in reducing page faults, as OBJECT VALUES has higher reductions than EXACT. Among all the evaluated workloads, EXACT performs better than OBJECT VALUES only on *Mandelbrot*, *Permute*, *Richards*, *Towers*, and *vanilla*. Strategy CODE+HEAP yields the largest page-fault reduction on all the evaluated workloads, 2.98× on average. The biggest improvement w.r.t. OBJECT VALUES is on *vanilla* (1.22×). Results show that partial matching is effective in reducing page faults, indicating that it successfully matches and orders objects that are not matched by exact matching due to divergences between image builds.

6.2.1 Comparison with Related Work. Fig. 11 shows the page-fault-reduction factors achieved by related work. Table 2 reports the geometric mean of the two best-performing ordering strategies CODE+HEAP (proposed by us) and CU+HEAP PATH (proposed by related work). We remark that HEAP PATH orders only the heap section (and hence can be compared with our strategy strategyheap) while CU+HEAP PATH orders both the code and heap sections (and hence can be compared with our strategy CODE+HEAP). Experimental results show that our HEAP strategy reduces an average of 1.67× page faults, while HEAP PATH reduces page faults by 1.21× on average. Moreover, our CODE+HEAP strategy outperforms CU+HEAP PATH, with reduction factors of 2.98× and 1.58×, respectively. Overall, our ordering strategies yield higher reduction factors than related work on all the workloads.

Table 2. Geomean of the page-fault reduction factors and execution-time speedups achieved by the best ordering strategies of this article (CODE+HEAP) and related work (CU+HEAP PATH), divided by benchmark suite.

Metric	Strategy	AWFY	Microservices	Overall
page-fault reduction	CODE+HEAP	3.14	2.72	2.98
	CU+HEAP PATH	1.65	1.47	1.58
execution-time speedup	CODE+HEAP	2.00	1.92	1.98
	CU+HEAP PATH	1.59	1.58	1.58

6.2.2 Evaluation on DaCapo and Renaissance. Fig. 13 shows the page-fault-reduction factors achieved by strategy CODE+HEAP on DaCapo and Renaissance. Experimental results show page-fault-reduction factors ranging from 1.23× to 2.61×, 1.81× on average. We notice that page fault reductions associated with these workloads are lower than the ones reported in Fig. 9 due to the different usage of libraries and static initializations.

6.3 Execution-Time Speedup

Fig. 10 reports the speedup achieved by the evaluated ordering strategies, which correlate with the page-fault reduction reported in §6.2, with an average factor of 1.30×, 1.30×, 1.75×, 1.80×, and 1.98× for CODE, HEAP, EXACT, OBJECT VALUES, and CODE+HEAP, respectively. This correlation

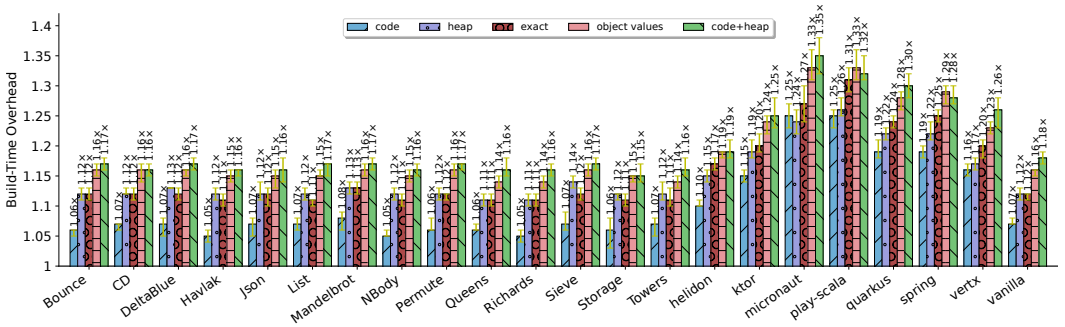


Fig. 15. Build-time overhead introduced by the ordering strategies.

indicates that page faults have a high impact on the execution time of Serverless and FaaS workloads. On AWFY, HEAP introduces higher speedups than CODE while on microservices CODE introduces higher speedups than HEAP. The maximum speedup is achieved by CODE+HEAP on workload *ktor* (2.52×), while EXACT introduces a slowdown of 0.51× on *play-scala*. We are currently investigating the reasons for this slowdown. Strategy OBJECT VALUES solves the slowdown on *play-scala* by introducing a speedup of 1.41×. Overall, CODE+HEAP introduces higher speedups with an average of 1.98×, confirming that code and heap ordering are synergistic.

6.3.1 Comparison with Related Work. Fig. 12 and Table 2 show the execution-time speedups achieved by the ordering strategies proposed by related work and the geometric means of the best performing strategies, respectively. Our HEAP strategy outperforms HEAP PATH, showing factors of 1.30× and 1.12×, respectively, while our CODE+HEAP yields better performance than CU+HEAP PATH, with factors 1.98× and 1.58×, respectively. Our CODE+HEAP strategy introduces higher speedups than related work on all the evaluated workloads, showing the effectiveness of the proposed approach.

6.3.2 Evaluation on DaCapo and Renaissance. Fig. 14 shows the execution-time speedups achieved by strategy CODE+HEAP on DaCapo and Renaissance. Experimental results show execution-time speedups ranging from 0.99× to 1.12×, 1.03× on average. As mentioned in §6.1, this is expected since DaCapo and Renaissance consist of async/batch, not startup-oriented, workloads that run for several seconds, where startup performance is not crucial and the startup time is very short w.r.t. steady-state execution time. We report the absolute execution-time numbers in Appendix E.

6.4 Build-Time Overhead

Fig. 15 reports the build-time overhead of our ordering strategies (lower is better). In the case of the baseline, the build time includes the time required to build the Native-Image binary by performing the steps represented by the white nodes in Fig. 2. In the case of optimized binaries, the build time includes also the steps represented by green and purple nodes in Fig. 2 and the time required to fetch and parse profiles stored in files. Experimental results show that heap ordering alone (HEAP) introduces an average build-time overhead of 1.14×, Strategies EXACT, OBJECT VALUES, and CODE+HEAP that order both code and heap introduce average build time factors of 1.15×, 1.18×, and 1.19×, respectively. Overall, the proposed approach for heap ordering incurs a moderate overhead. Moreover, object values and partial matching do not introduce significantly higher slowdowns w.r.t. exact matching alone—CODE+HEAP introduces an average build-time overhead of 1.03× w.r.t. EXACT. The build-time overhead does not depend on the number of ordered objects (as later shown in §6.5) but rather on the object-matching process. We note that the overhead of partial-path matching depends on the heap-snapshot size and the number of exactly-unmatched

Table 4. Ordering statistics for the evaluated strategies on microservice frameworks.

Strategy	Workload	# Profiled IDs	# Exactly-Matched Ordered Objects	# Partially-Matched Ordered Objects	Strategy	Workload	# Profiled IDs	# Exactly-Matched Ordered Objects	# Partially-Matched Ordered Objects
HEAP	*helidon*	7588	14318	174	EXACT	*helidon*	5978	59896	0
	ktor	9448	12470	182		*ktor*	7698	96059	0
	micronaut	19938	31205	298		*micronaut*	17446	111940	0
	play-scala	37622	50805	562		*play-scala*	26885	156328	0
	quarkus	7003	11330	497		*quarkus*	5656	64817	0
	spring	34602	50931	1265		*spring*	27179	148236	0
	vertx	7718	13291	295		*vertx*	6406	77238	0
	vanilla	2791	6584	496		*vanilla*	2446	19966	0
OBJECT VALUES	*helidon*	7588	14619	0	CODE+HEAP	*helidon*	7492	14222	146
	ktor	9447	12455	0		*ktor*	9445	12494	132
	micronaut	19941	31045	0		*micronaut*	19933	31013	319
	play-scala	33112	43534	0		*play-scala*	33156	43485	495
	quarkus	6992	11357	0		*quarkus*	6989	11328	476
	spring	34704	51641	0		*spring*	34756	50779	1171
	vertx	7710	13330	0		*vertx*	7723	12705	975
	vanilla	2791	6350	0		*vanilla*	2788	6313	338

objects (the latter is small for the evaluated workloads). In the case of a high number of exactly-unmatched objects, it is possible to create a suffix tree to find a match while tracking the best score. Each partial-path match would have the complexity of the longest path in the image heap. AWFY benchmarks are associated with lower overheads than microservices, since the latter include more code and larger heap snapshots. We remark that the build time is often not critical—build time is not what would eventually break the SLA. For space reasons, we report the build-time overhead on DaCapo and Renaissance in Appendix F.

6.5 Object-Ordering Statistics

Here, we analyze heap-ordering statistics collected on microservices, to explain the effects of the evaluated ordering strategies. We focus on microservices since these workloads include bigger heap snapshots, which better illustrate the effects of the evaluated strategies.

Table 3. Object statistics for the evaluated strategies on microservice frameworks.

Workload	# Objects in Heap Snapshot	# Accessed Objects	% Accessed Objects
helidon	236130	8237	3.49
ktor	331766	10144	3.06
micronaut	394170	24680	6.26
play-scala	438096	35145	8.02
quarkus	277303	8494	3.06
spring	410939	38226	9.30
vertx	312050	8314	2.66
vanilla	138452	2997	2.16

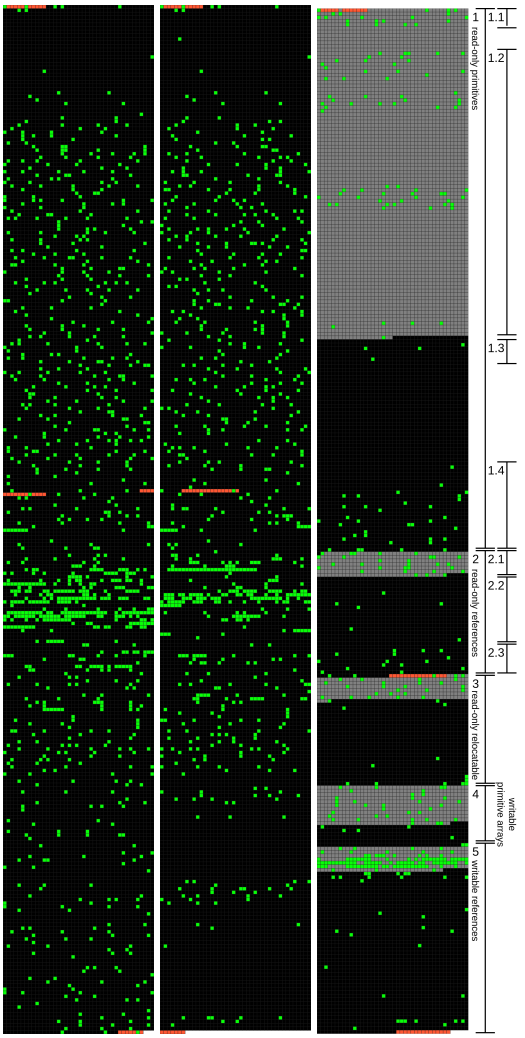
Table 3 reports the number of objects in the heap snapshot and the amount of accessed objects per workload. These numbers do not change based on the strategy. As the table shows, only a small percentage of the objects in the snapshot is accessed at runtime (5.4% on average), which indicates the importance of ordering/compacting objects according to their access order to reduce page faults.

For each heap-ordering strategy, Table 4 reports statistics related to object-access profiling, object matching, and ordering. In particular, the table reports the number of profiled IDs (i.e., the accessed-object count according to the heap-ordering profiles collected during the execution of the instrumented image and used at optimized-image build time), of exactly-matched ordered objects and of partially-matched-ordered objects in the optimized image. The values are the arithmetic means across all optimized-image builds of the respective workload.

Profiling and Collisions. The table shows that the number of profiled IDs varies among the strategies. The reason is that different strategies lead to different numbers of collisions—even in the case where the application accesses the same objects when using different strategies, each strategy assigns a different number of objects to the same ID (and hence the same CAHP as described in §4.1.2). The lower number of profiled IDs associated with EXACT shows that many objects in this strategy have the same IDs. This is expected since, without array indices and object values in CAHPs, all the objects of the same type stored in the same array obtain the same ID. Object values

mitigate this issue as demonstrated by the statistics for OBJECT VALUES. The HEAP strategy usually profiles slightly more IDs than CODE+HEAP, i.e., the corresponding strategy that also employs code ordering. The number of profiled IDs is lower than the one of accessed objects reported in Table 3 for all evaluated strategies, indicating that collisions are present for every strategy.

Ordering and Collisions. The number of exactly-matched ordered objects is higher than the number of profiled objects, which is expected—some optimized heap-snapshot objects that are not accessed have the same IDs as objects that were accessed. Furthermore, the exactly-matched ordered-objects column confirms that EXACT assigns the same ID to considerably more objects than other strategies—despite the low number of profiled IDs, we order more exactly-matched objects than other strategies. The number of partially-matched ordered objects is similar for both the strategies that exploit partial matching. We can notice that partial matching orders only a few objects. However, the page-fault reduction factors reported in §6.2 indicate that these objects are correctly matched, ordered, and then accessed at runtime, showing the effectiveness of partial matching. The sum of the exactly- and partially-matched ordered objects is higher than the number of accessed objects for all workloads and strategies. This indicates that every strategy orders objects that are not accessed at runtime due to ID collisions. However, the relative error, i.e., the number of objects that we order and that are not accessed at runtime, is small w.r.t. the total number of objects stored in the heap snapshot. Despite this, the ordering techniques overall improve performance by reducing page faults, and this confirms the results from §6.2 and §6.3.



(a) Not optimized. (b) Related work. (c) Our strategy.

Fig. 16. Heap section of the *play-scala* workload and the corresponding page faults.

7 Optimized-Heap-Snapshot Visualization and Analysis

This section provides a visualization and an analysis of the heap section of an optimized image. To perform the analysis, we exploit the debugging information collected following the methodology described in §4. In particular, we identify the objects causing page faults and we inspect their CAHPs. The CAHPs allow us to determine the type of the objects as well as their semantics (e.g., we identify interned strings, objects storing metadata, and objects created by application code). During development, we employed this information and analysis methodology to determine what object-context information to include in the CAHPs and tune exact and partial matching.

Fig. 16a, Fig. 16b, and Fig. 16c show the .svm_heap section of a regular image (generated without applying any strategy proposed by related work), an image generated by related work [4], and an

optimized image generated by our approach for workload *play-scala*, respectively. We use *play-scala* since it is the workload with the highest number of objects in the heap snapshot (as shown in Table 3) and shows interesting insights. The figures use the same format of Fig. 1a and 1b (explained in §1), with the addition of the gray cells to indicate pages that have not been paged in by the OS and contain objects that have been ordered. Since a small percentage of the objects in the heap snapshot is accessed at runtime, a small percentage of objects is ordered (as shown in §6.5)—we do not expect the heap to be predominantly gray. The page size is 4KB. Page faults (green cells) surrounded by gray cells indicate page faults caused by ordered objects, while page faults surrounded by black cells indicate page faults caused by unordered objects. Next to the heap section of the optimized image, we report numbers to identify segments of that heap section for the sake of explanation. First, Fig. 16a and 16b show that the regular image and the one generated by related work cause page faults throughout the heap section, while Fig. 16c shows fewer more-localized page faults—accessed objects are compacted in fewer pages, leading to performance improvements. We point out that the ordered objects (identified by the gray regions) are not contiguous. This is because Native Image splits the heap section in several segments (Seg.), each of them containing objects with different properties. Seg. 1 contains read-only primitive arrays, Seg. 2 contains read-only references, Seg. 3 contains read-only relocatable objects, while Seg. 4 and 5 contain writable primitive arrays and writable references, respectively. In the following text, we analyze each segment separately.

We divide Seg. 1 into four segments numbered from 1.1 to 1.4. Seg. 1.1 and 1.2 are ordered and contain mostly byte arrays associated with interned strings and large byte arrays that contain Native Image metadata, respectively. In particular, Seg. 1.2 contains four large arrays containing method and reflection metadata that are used to e.g. create stack traces for exceptions. This explains the large number of gray pages in this segment—this multi-page array is accessed, but only a subset of the entires are accessed (corresponding to methods for which stack traces were created in the image execution). Seg. 1.3 and 1.4 contain mostly byte arrays that have not been ordered and are associated with strings and interned strings, respectively. The reason for the page faults associated with the byte arrays of the interned strings (i.e., why we do not order several interned strings) is that Native Image stores interned strings in a sorted array and performs a binary search at runtime when invoking the method String.intern(). To order these interned strings, the instrumented image should access (and profile) the strings that will be later accessed by the binary search when executing the optimized image. However, this is not the case. The instrumented and optimized images contain slightly different interned strings. For example, the instrumented image contains interned strings related to the instrumentation code, such as the names of the instrumentation classes. These strings alter the access pattern of the binary search so that the instrumented-image execution accesses different string byte arrays w.r.t. the string byte arrays that would have been accessed by a regular or an optimized image. Different accesses lead to inaccuracies in the profiles and consequently in the ordering, limiting the number of objects that are reordered, and hence the page-fault reduction.

Seg. 2 contains mostly java.lang.String and java.lang.Class objects. Seg. 2.1 contains correctly ordered objects, Seg. 2.2 contains primarily unordered (non-interned) strings, while Seg. 2.3 contains many unordered interned strings. The cause for page faults in Seg. 2.3 is similar to that of the page faults in Seg. 1.4, i.e., the binary search accesses both the java.lang.String wrapper object and the byte array that this object points to. Seg. 3 contains mainly java.lang.Class objects. The gray section in Seg. 4 is dominated by a large byte array containing reflection metadata. Seg. 5 contains most of the objects allocated by the application, and shows the effectiveness of our ID generation—many page faults are contiguous, indicating that many of the accessed objects are packed together. A gap between contiguous page faults is likely caused by either (1) a large object spanning multiple pages or (2) many objects associated with the same ID, placed one after the other. In the former, the application accesses only some elements of this large object, while in the latter, the application

accesses only some of the objects associated with the same ID. Page faults outside of the gray sections are caused by objects that have not been matched and ordered due to inaccuracies.

8 Related Work

8.1 Optimizing Heap Ordering

To the best of our knowledge, Basso et al. [4] propose the only binary-reordering technique for Native Image. The main novelty of their work is presenting a new approach to map objects in two different binaries built from the same application, where significant divergences in the object graph are expected. As already discussed in §2, our approach improves the heap-ordering strategies proposed by the above work, which result in limited startup speedups due to their poor capabilities in mapping semantically equivalent objects across binaries (as shown in §6). Moreover, their strategies do not perform partial matching and do not produce metadata that can be exploited for debugging and visualization purposes, unlike our work (as detailed in §4.4 and §7). We are not aware of any other work that proposes strategies to map semantically equivalent objects across heap snapshots produced by different compilations.

While improving heap ordering is a common technique to increase performance, related work apply it to optimize runtime memory allocations [8, 14, 17, 22]; in contrast, our goal is to improve the ordering of objects stored in a heap snapshot embedded in a binary. Other approaches optimize the data-layout to improve locality [18, 42, 49]. Finally, a few work exploit PGO (as we do) to optimize the heap layout [28, 31, 44]. However, none of these approaches proposes techniques for matching corresponding objects across compilations, which is our goal.

Finally, several strategies proposed by related work attempt to compute a one-to-one node mapping between two graphs [7, 9, 13, 16, 24, 53] Unfortunately, none of these strategies is designed to consider the peculiarities of the heap-object graphs and the divergences across builds. For example, the two nodes corresponding to the semantically equivalent objects may be connected differently in the two graphs. In our work, we employ domain knowledge to guide the matching algorithm, showing that matching can be effectively performed not only by considering the graph structure but also the object content (i.e., constructing CAHPs using elements of type Object Value).

8.2 Improving Startup Performance

Optimizing the binary layout to reduce page faults has been explored in recent research [20, 23]. However, such techniques focus on code reordering; instead, our approach aims at improving object reordering. It should be noted that the above work attempt to reduce page faults by reducing the binary size. Such strategies are only partly suitable for Native Image, where the majority of the binary (typically ~60%) is occupied by the heap snapshot.

Due to the growing diffusion of Serverless and FaaS, modern research is increasingly prioritizing the optimization of startup performance, focusing both on improving the execution of interpreted code [3, 38] and reducing the startup time of the dynamic compiler [2, 27, 36, 52]. Moreover, modern implementations of managed runtimes offer ways to pre-initialize the execution context [50, 54], aiming at shortening the initialization time of the runtime. Finally, some work specifically focus on optimizing startup performance of Serverless and FaaS workloads [1, 25]. These work are complimentary to our approach and show that optimizing startup performance is a timely and important research topic.

9 Discussion

Alternative and Complementary Approaches. As an alternative approach, instead of building an instrumented and an optimized binary, one could 1) profile the execution of one binary to track

the executed code and accessed objects, and 2) use the collected profiles to re-layout the same binary. This strategy removes the need to perform object-matching, potentially leading to optimal locality and startup performance. However, this alternative approach incurs different challenges. First, when building the binary, one needs to dump additional metadata that is later required to relocate objects, patch references between objects and references embedded in the code. Second, to profile the binary without incurring prohibitive overhead, one would need hardware support for tracing accessed memory addresses. We are not aware of such hardware capabilities. Our approach reduces overhead by instrumenting only memory accesses corresponding to object accesses (Appendix A.1). We consider exploring this alternative approach an interesting future work.

To reduce the impact of page faults, one could employ prefetching strategies to preemptively load the pages containing the profiled code and objects into memory. Since our approach reduces the number of pages to (pre)fetch by compacting the accessed objects in fewer contiguous pages, we consider prefetching complementary to our approach.

Limitations. The main limitation of our approach is that we assume that most of the semantically different objects are not associated with the same CAHP, which may introduce inaccuracies. The instrumentation code introduces some degree of inaccuracy in object matching as well. In addition to the perturbation introduced by the instrumentation code on the binary search on interned strings (explained in §7), instrumentation code may allocate objects that are not present in the optimized image, such as metadata associated with instrumentation classes, which is stored in maps included in the heap snapshot. The inclusion of this metadata may lead to collisions in maps allocated in the instrumented image and hence to the allocation of additional buckets (used internally by the map) that are not present in the optimized image. Additional map nodes in turn lead to the computation of different CAHPs for semantically equivalent objects between the instrumented and optimized image. Our approach mitigates this issue by performing partial path matching. The divergence of heap snapshots is not a direct limitation of our work, but a limitation of any object-matching technique that relies on Native Image and implements instrumentation logic in Java bytecode. Furthermore, our object-ordering approach may introduce false positives (i.e. objects that are reordered, but not accessed in the optimized execution), as discussed in §6.5. Despite the presence of false positives, experimental results show significant page-fault reductions and considerable performance improvements w.r.t. to the default Native Image implementation and related work.

10 Concluding Remarks

In this article, we propose a novel approach to improve the mapping between semantically equivalent objects of two different binaries generated by GraalVM Native Image on the same application. Our approach aims at improving the efficiency (in terms of page-fault and startup-time reduction) of profile-guided approaches that reorder objects in the heap snapshot of an optimized binary based on the order in which objects are first accessed, obtained by profiling an instrumented binary of the same application. We propose to associate each object to be stored in the heap snapshot with a Context-Augmented Heap Path (CAHP), and consider objects with the same CAHP across different binaries as semantically equivalent. Moreover, since non-determinism in the image-build process may lead to different CAHPs for semantically equivalent objects, we present an approach that further improves the accuracy of object-matching, finding for each unmatched CAHP in the optimized binary, the most similar CAHP in the instrumented binary, associating the two objects. We integrate our approach in Native Image and evaluate it on FaaS benchmarks as well as on widely-used microservice frameworks, obtaining an average reduction of page faults of 2.98× and an average startup speedup of 1.98× w.r.t. the original Native Image implementation.

Data-Availability Statement

The artifact [5] 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. The artifact contains a set of tools/scripts that can be used to execute the workloads, collect, process, and plot page-fault and performance measurements to replicate the evaluation presented in the article. The artifact also contains the complete pre-collected measurements used to generate the original figures of the article.

Acknowledgments

This work has been supported by Oracle ("Dynamic Regression-Detection and Compiler Analysis Framework for GraalVM", ID 5161), by the Swiss National Science Foundation, and by the USI FIR project "Understanding and Mitigating Performance Variability on Managed Runtimes". 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] Amazon Web Services - Labs. 2025. LLRT GitHub Repository. https://github.com/awslabs/llrt
[2] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *OOPSLA*. 297–311. doi:10.1145/1094811.1094835
[3] Matteo Basso, Daniele Bonetta, and Walter Binder. 2023. Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance. In *GPCE*. 1–13. doi:10.1145/3624007.3624050
[4] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2025. Improving Native-Image Startup Performance. In *CGO*. 689–703.
[5] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2025. Artifact associated to the paper "Heap-Snapshot Matching and Ordering using CAHPs: A Context-Augmented Heap-Path Representation for Exact and Partial Path Matching using Prefix Trees" published in OOPSLA'25. doi:10.5281/zenodo.16522289 artifact.
[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis *(OOPSLA'06)*. 169–190.
[7] Tibério S. Caetano, Julian J. McAuley, Li Cheng, Quoc V. Le, and Alex J. Smola. 2009. Learning Graph Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 6 (2009), 1048–1058.
[8] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS*. 139–149. doi:10.1145/291006.291036
[9] Minsu Cho, Jungmin Lee, and Kyoung Mu Lee. 2010. Reweighted Random Walks for Graph Matching *(ECCV'10)*. 492–505.
[10] Du, Dong and Yu, Tianyi and Xia, Yubin and Zang, Binyu and Yan, Guanglu and Qin, Chenggang and Wu, Qixuan and Chen, Haibo. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting *(ASPLOS'20)*. 467–481.
[11] 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. doi:10.1145/2542142.2542143
[12] Eclipse. 2025. Vert.x Framework. https://vertx.io/
[13] Amir Egozi, Yosi Keller, and Hugo Guterman. 2013. A Probabilistic Approach to Spectral Graph Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1 (2013), 18–27.
[14] Yi Feng and Emery D. Berger. 2005. A Locality-improving Dynamic Memory Allocator. In *MSP*. 68–77. doi:10.1145/1111583.1111594
[15] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS*. 386–400. doi:10.1145/3445814.3446757
[16] S. Gold and A. Rangarajan. 1996. A graduated Assignment Algorithm for Graph Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18, 4 (1996), 377–388.
[17] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *PLDI*. 177–186. doi:10.1145/173262.155107

[18] Christopher Haine, Olivier Aumage, and Denis Barthou. 2017. Rewriting System for Profile-Guided Data Layout Transformations on Binaries. In *Euro-Par*. 260–272. doi:10.1007/978-3-319-64203-1_19
[19] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *PASTE*. 54–61.
[20] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing Function Layout for Mobile Applications. In *LCTES*. 52–63. doi:10.1145/3589610.3596277
[21] JetBrains. 2025. Ktor Framework. https://ktor.io/
[22] Alin Jula and Lawrence Rauchwerger. 2009. Two Memory Allocators that Use Hints to Improve Locality. In *ISMM*. 109–118. doi:10.1145/1542431.1542447
[23] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-guided Size Optimization for Native Mobile Applications. In *CC*. 243–253. doi:10.1145/3497776.3517764
[24] Marius Leordeanu and Martial Hebert. 2009. Unsupervised Learning for Graph Matching *(CVPR'09)*. 864–871.
[25] 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. doi:10.1145/3585007
[26] LLVM Project. 2025. Benchmarking Tips. https://llvm.org/docs/Benchmarking.html
[27] 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. doi:10.1145/3132190.3132210
[28] Chaitanya Mamatha Ananda, Rajiv Gupta, Sriraman Tallam, Han Shen, and Xinliang David Li. 2025. PreFix: Optimizing the Performance of Heap-Intensive Applications. In *CGO*. 405–417. https://doi.org/10.1145/3696443.3708960
[29] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *DLS*. 120–131. doi:10.1145/2989225.2989232
[30] Micronaut Foundation. 2024. Micronaut Framework. https://micronaut.io/
[31] 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. *ACM Trans. Embed. Comput. Syst.* 22, 1, Article 2 (2022). https://doi.org/10.1145/3527853
[32] Oracle and/or its affiliates. 2021. GraalVM: Native Image. https://www.graalvm.org/jdk21/reference-manual/native-image/
[33] Oracle and/or its affiliates. 2025. DaCapoBenchmarkSuite. https://github.com/oracle/graal/blob/1ff637ab400c9098d2c06606d646e4733e8af9a5/java-benchmarks/mx.java-benchmarks/mx_java_benchmarks.py#L1000
[34] Oracle and/or its affiliates. 2025. Helidon Framework. https://helidon.io/
[35] Oracle and/or its affiliates. 2025. RenaissanceBenchmarkSuite. https://github.com/oracle/graal/blob/1ff637ab400c9098d2c06606d646e4733e8af9a5/java-benchmarks/mx.java-benchmarks/mx_java_benchmarks.py#L1988
[36] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *CGO*. 340–350. doi:10.1109/CGO51591.2021.9370314
[37] Play Framework. 2025. Play Framework. https://spring.io/
[38] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *POPL*. 322–332. doi:10.1145/199448.199526
[39] 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. doi:10.1109/CGO.2019.8661171
[40] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM *(PLDI'19')*. 31–47.
[41] Red Hat. 2025. Quarkus. https://quarkus.io/
[42] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An Efficient Profile-analysis Framework for Data-layout Optimizations. In *POPL*. 140–153. doi:10.1145/565816.503287
[43] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *CC*. 126–137. doi:10.1007/3-540-36579-6_10
[44] Joe Savage and Timothy M. Jones. 2020. HALO: Post-link Heap-layout Optimisation. In *CGO*. 94–106. doi:10.1145/3368826.3377914
[45] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *CGO*. 165–174. doi:10.1145/2581122.2544157
[46] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution *(HPCA'23)*. 814–827.
[47] The Kernel Development Community. 2025. Documentation for /proc/sys/vm/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html?highlight=drop_caches#drop-caches

[48] VMware Tanzu. 2025. Spring Framework. https://spring.io/
[49] 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. doi:10.1109/AEMCSE55572.2022.00055
[50] 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. doi:10.1145/3360610
[51] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. In *PLDI*. 990–101. doi:10.1145/3656417
[52] 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. doi:10.1145/3276494
[53] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. 2016. A Short Survey of Recent Advances in Graph Matching *(ICMR'16)*. 167–174.
[54] Yang Guo. 2015. Custom Startup Snapshots. https://www.v8.dev
[55] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets *(HotCloud'10)*. 10 pages.

Received 2025-03-26; accepted 2025-08-12