Improving Native-Image Startup Performance

Matteo Basso[*], **Aleksandar Prokopec**[†], **Andrea Rosà**[*], **Walter Binder**[*]

[*] Università della Svizzera italiana, Switzerland
[†] Oracle Labs

CGO'25
March 5, 2025

Introduction - Serverless and FaaS

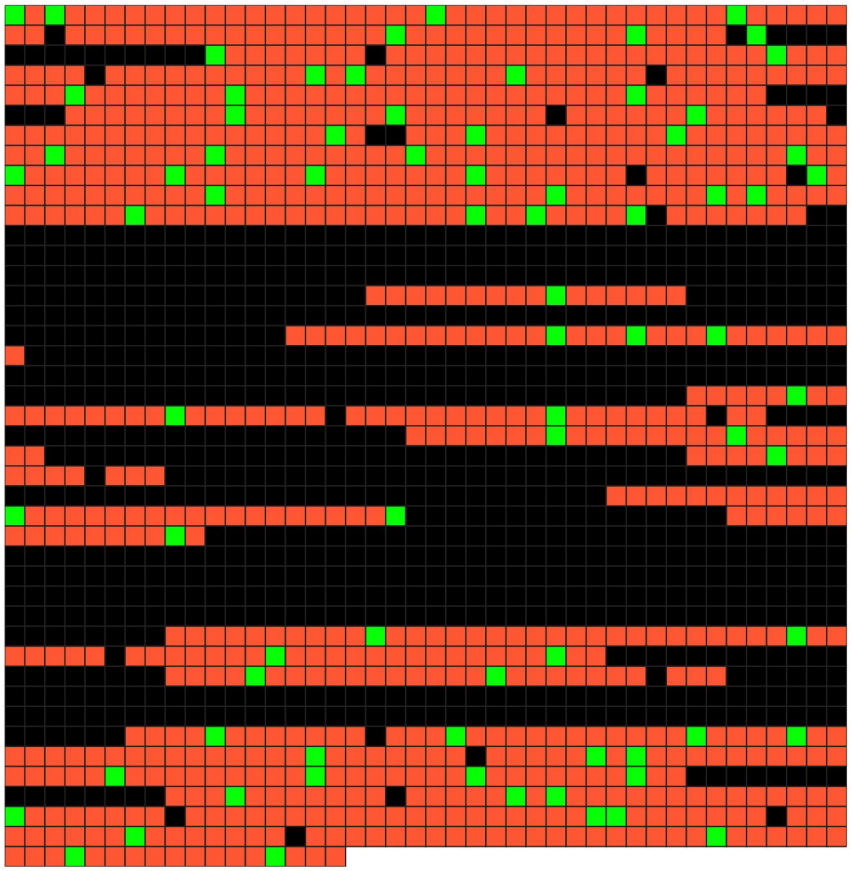
- Short-running workloads
  - Often not optimized by Just-In-Time (JIT) compilation due to its overheads in the startup
- The first execution of the workload on a machine requires the initialization of the execution environment
  - The code is either fully downloaded or incrementally downloaded using a Network File System (NFS) upon page faults
  - Initialization may take place several times
    - To avoid wasting resources, the service typically retains the execution environment only for a certain period of time
      - The idle program is removed
    - Optimization is crucial to lower costs and maximize throughput

Goal and Focus

- Our goal is improving the locality of the executed code and accessed objects to reduce page faults and hence I/O traffic
- We focus on GraalVM Native Image [1] which allows creating a binary file from a Java application
  - Machine code emitted by leveraging the Graal compiler as an Ahead-of-Time (AOT) compiler
  - Executed without instantiating a Java Virtual Machine (JVM)
  - The binary contains not only the code to be executed, but also a snapshot of the pre-initialized heap memory

[1] Wimmer et al., “Initialize Once, Start Fast: Application Initialization at Build Time”. OOPSLA’19.

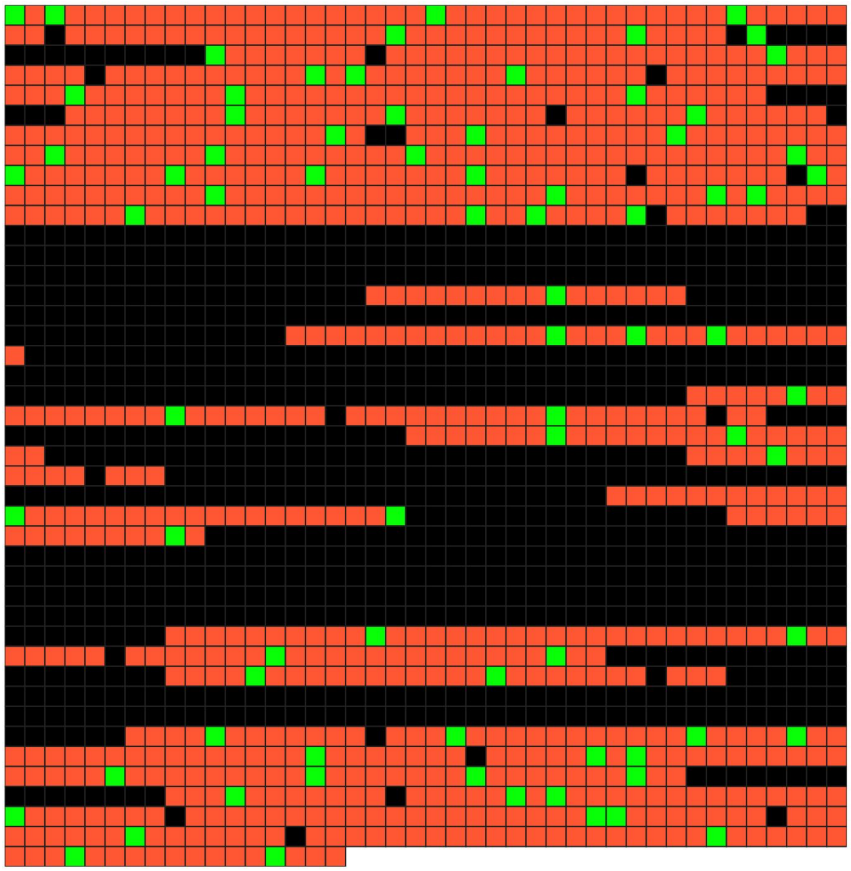
Visualization of the Code Section (1)



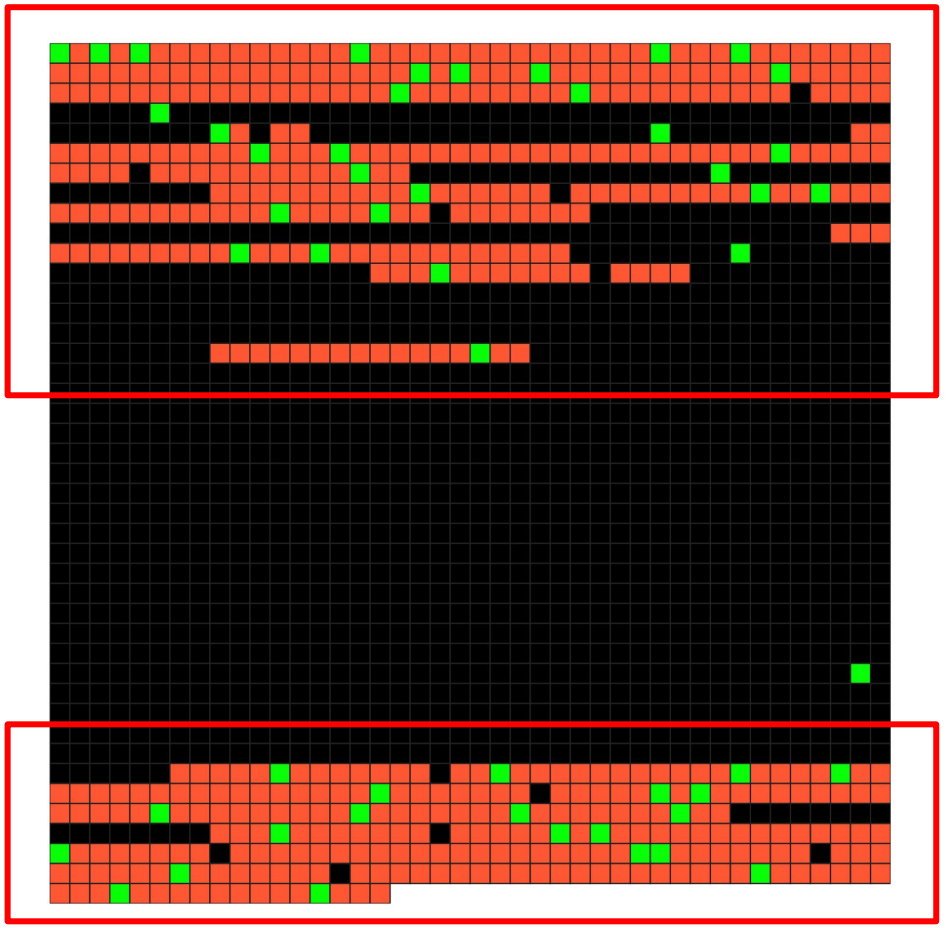
(a) Regular binary

- Code section of a Native-Image binary
- Each cell represents a page
  - **Green**: caused page faults
  - **Red**: prepaged by the OS
  - **Black**: not fetched

Visualization of the Code Section (2)



(a) Regular binary



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

Contributions

- We propose a profile-guided methodology to improve the startup time of Native Image binaries by reordering the code and the heap-snapshot sections of the binary
  - We first generate an instrumented binary of the program to collect a method invocation trace and an object access trace
  - Using the trace, we create a second, profile-driven optimized binary where used methods and objects are placed in contiguous areas
- We design two code-ordering strategies and three heap-ordering strategies
  - Divergences in inlining decisions between builds of the same program
  - Matching objects from a profile against the objects in the profile-guided build
- We evaluate our implementation showing that it reduces page faults and improves runtime performance by 1.61× and 1.59×, respectively

Background & Challenges (1)

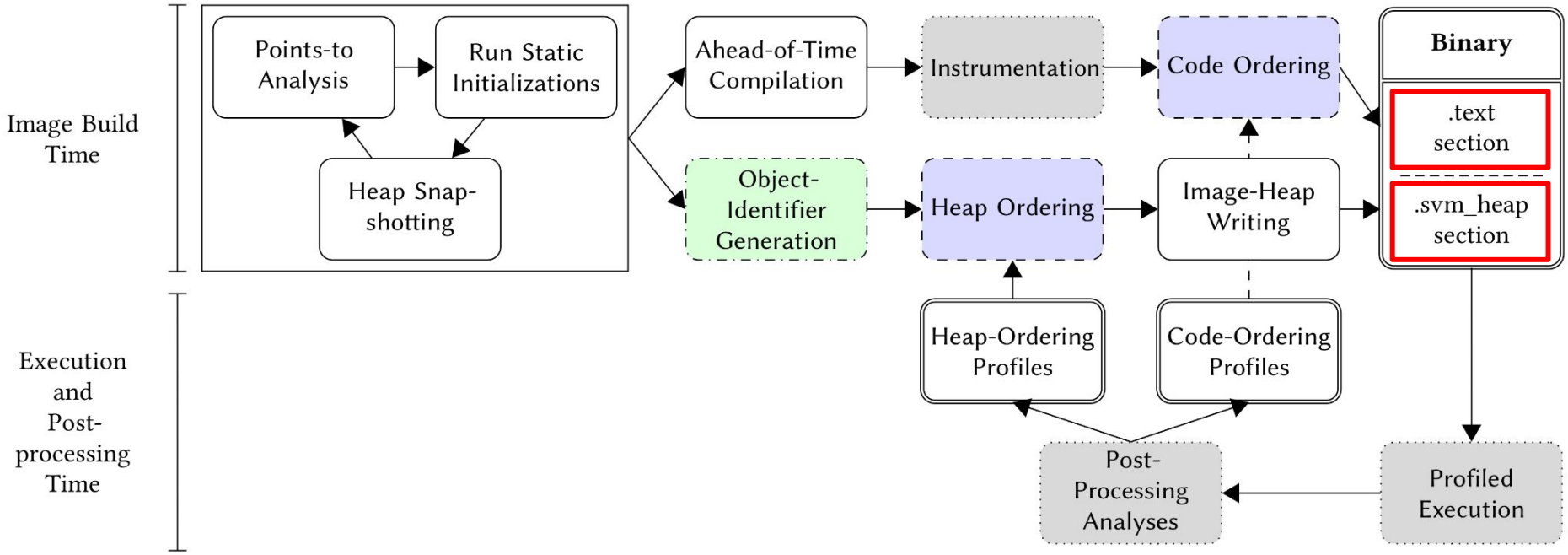
- The Graal compiler performs transformations and optimizations on a portion of code provided as input, called compilation unit (CU)
  - A CU consists of a root method and the methods that were inlined into the CU
- Native Image employs a points-to analysis to decide which code is reachable (and hence must be included in the binary) and saturation [1] to improve compilation speed
  - Binaries include more code than reachable or executed at runtime
- The inclusion of seemingly unrelated code (and instrumentation code) in the binary may significantly impact (code-size driven) inlining decisions
  - Different builds contain different CUs, causing divergences between (the instrumented and) the regular images
  - Inaccuracies in the profiles and hence in the profile-driven images

[1] Wimmer et al., “Scaling Type-Based Points-to Analysis with Saturation”. PLDI’24.

Background & Challenges (2)

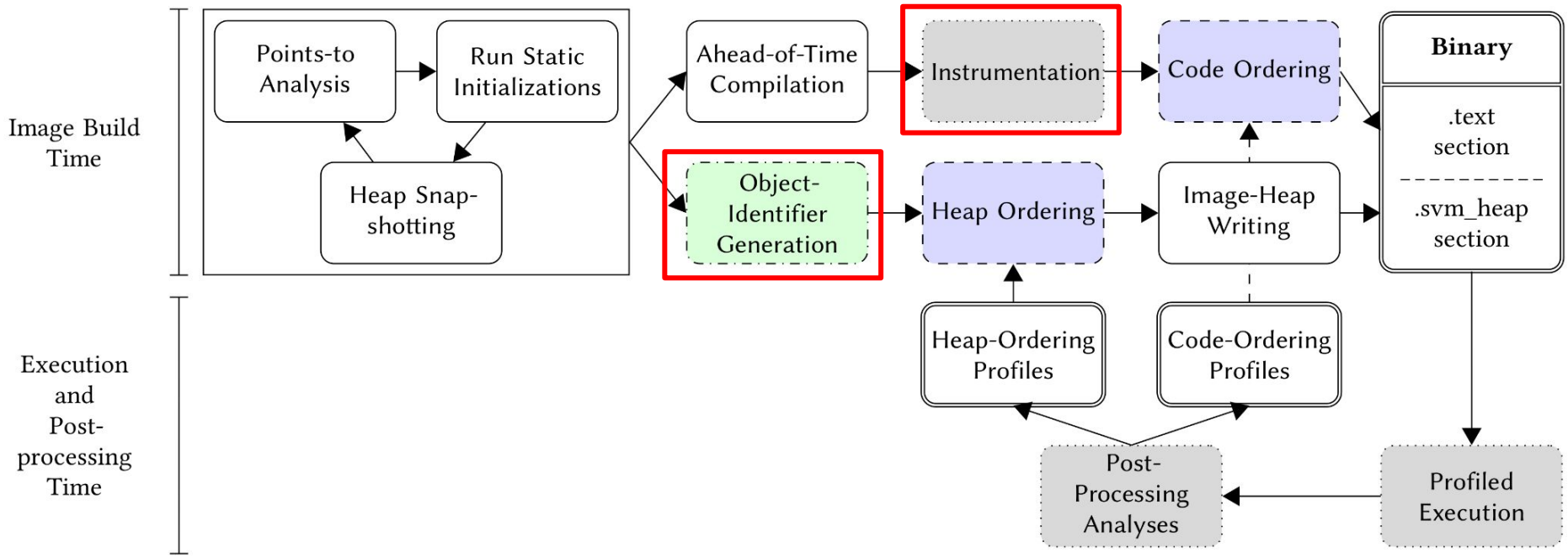
- The heap snapshot is obtained after concurrently executing the static initializers of the classes that are deemed to be reachable in the startup process of the VM
  - Heap snapshots typically differ across compilations
    - For example, due to different inlining decision that affect Partial Escape Analysis
- While CUs can be mapped across builds using the signature of their root methods, objects do not offer APIs that allow mapping their identities across builds
  - For example, the hash computed by `System.identityHashCode` on the semantically same object most likely differs across builds
  - It is challenging to match the object-access trace entries with the heap-snapshot objects of the optimized binary

Profile-guided Binary Reordering (1)



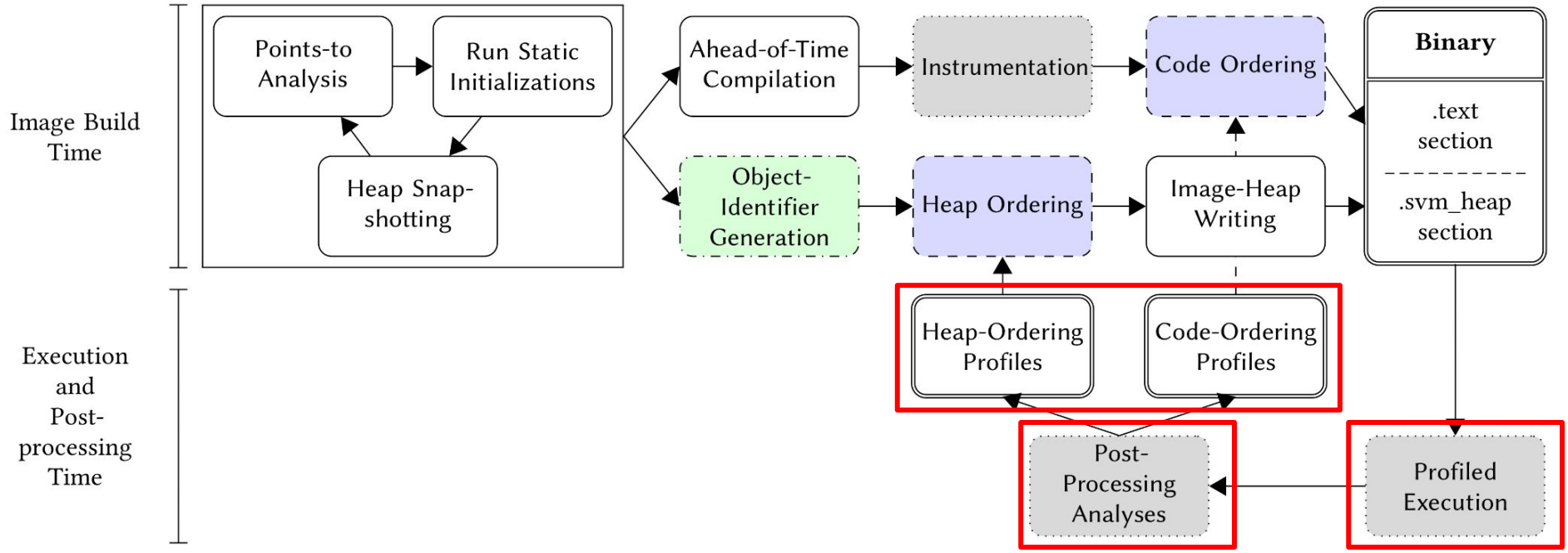
- Our goal is to improve the existing profiles collected by instrumented Native-Image binaries, and use the augmented profiles to generate an optimized binary
  - Order CUs in the .text section
  - Order objects in the .svm_heap section

Profile-guided Binary Reordering (2)



- Instrumented-image build time
  - We perform instrumentation ahead-of-time
  - To map object identities across builds, we generate object identifiers before writing the image heap

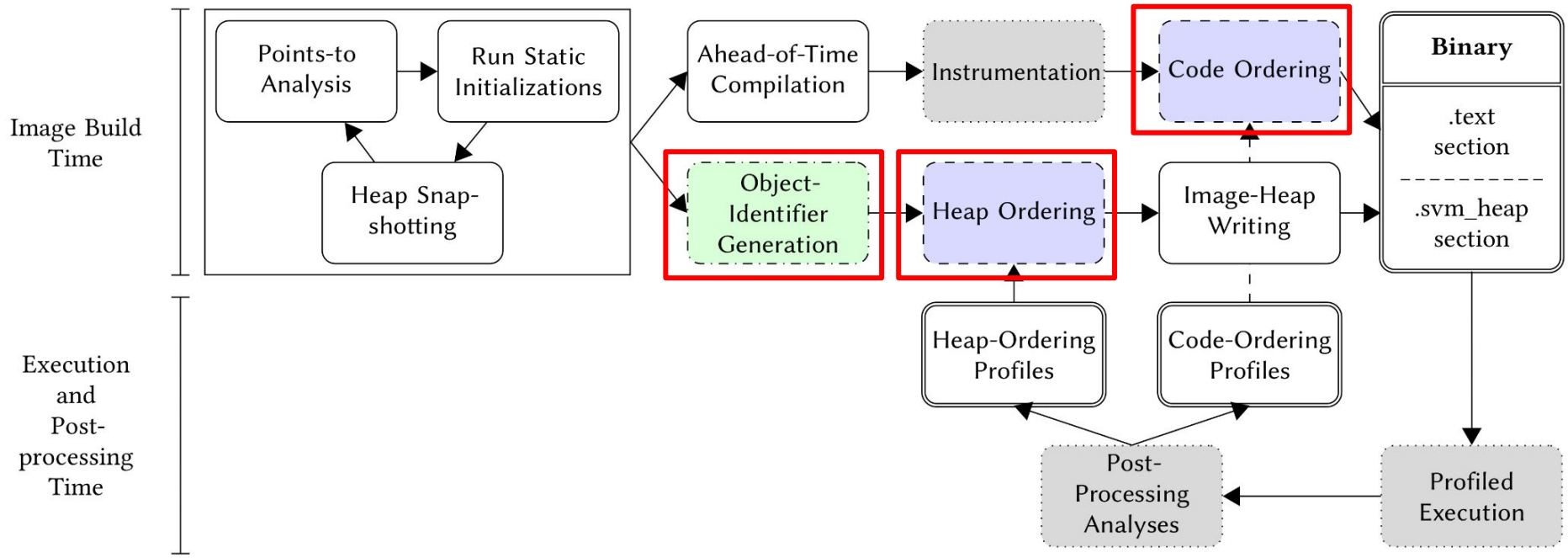
Profile-guided Binary Reordering (3)



➢ Instrumented-image execution time

- We collect invocation trace and an object access trace
- We post-process the traces to generate profiles that can be exploited by the optimized-image build process

Profile-guided Binary Reordering (4)



➢ Optimized-image build time

- We order code according to the ordering reported in the code-ordering profiles
- We order objects according to the ordering reported in the heap-ordering profiles
  - We match the identifiers in the profiles with the newly computed identifiers

Code Ordering (1)

- Reducing code-related page faults
  - Given two methods: A and B
  - If the first invocation of method A appears in the trace before the first invocation of method B, method A should be stored in the binary before method B
- In practice:
  - The binary contains several copies of the same method due to code duplication and inlining
  - Copies may be different across images
  - Given a choice of CUs, it is challenging to determine the optimal ordering

Code Ordering (2)

- We implement and evaluate two code-ordering heuristics:
  - **CU ordering**: we order the CUs based on the invocation order of the root methods
  - **Method ordering**: we order the CUs based on the invocation order of all the methods

Heap Ordering

- Heap-ordering strategies compute 64-bit object identifiers (IDs) to match the object-access trace entries with the heap-snapshot objects of the optimized binary as accurately as possible
  - **Incremental ID**: leverages the heap object graph traversal order
    - Assigns incremental IDs to object instances in object encounter order when traversing the heap object graph
  - **Structural Hash**: leverages the objects content
    - Analyzes the object structures and hashes the content of all their fields
  - **Heap Path**: leverages the inclusion reason in the heap snapshot
    - Hashes the first path in the heap object graph (starting from a root) to that object found by Native Image

Profiling

- Tracing profiler
  - Per-thread sequence of executed events
  - Compiler IR-level instrumentation
    - Increases profile accuracy and lowers perturbation on compiler optimizations [1]
  - Code ordering
    - CU/method entry events
  - Heap ordering
    - All the identifiers of the accessed Java objects (field/array accesses, monitor acquisitions, etc)

[1] Basso et al., “Optimization-Aware Compiler-Level Event Profiling”. TOPLAS, 2023.

Evaluation

- Performance evaluation of our implementation
  - On the “Are We Fast Yet?” (AWFY) benchmark suite [1]
    - To evaluate the improvements on the FaaS model
  - On popular microservice frameworks: micronaut [2], quarkus [3], spring [4]
    - To evaluate the improvements on the serveless model when employing long-running processes

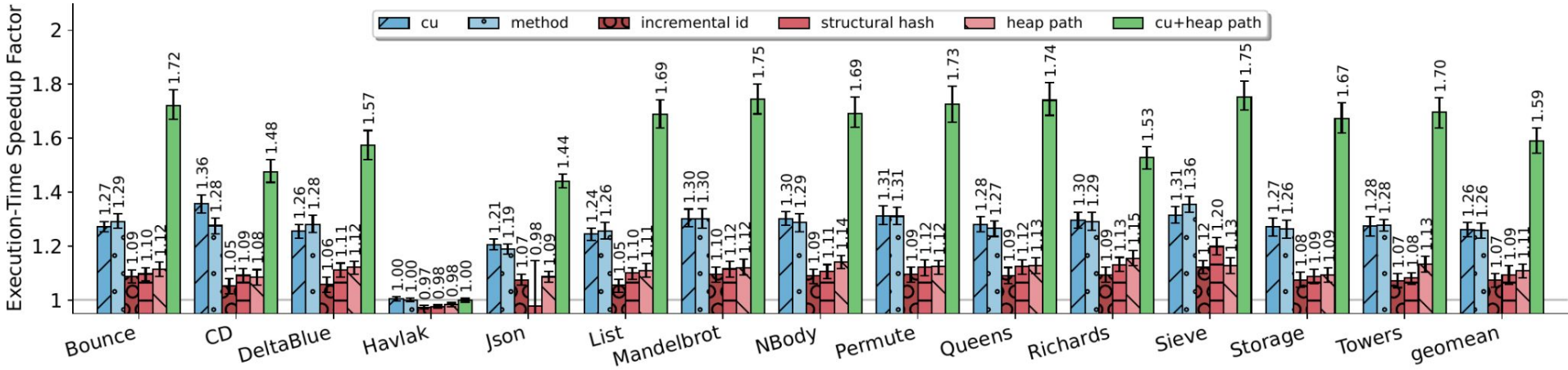
[1] Marr et al., “Crosslanguage Compiler Benchmarking: Are We Fast Yet?”. DLS’16.

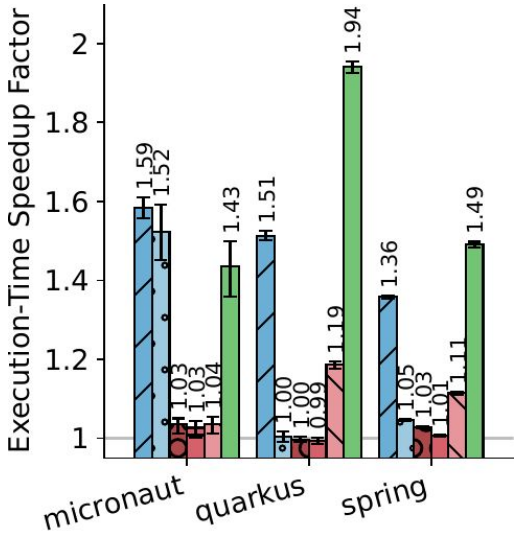
[2] https:// micronaut.io/

[3] https://quarkus.io/

[4] https://spring.io/

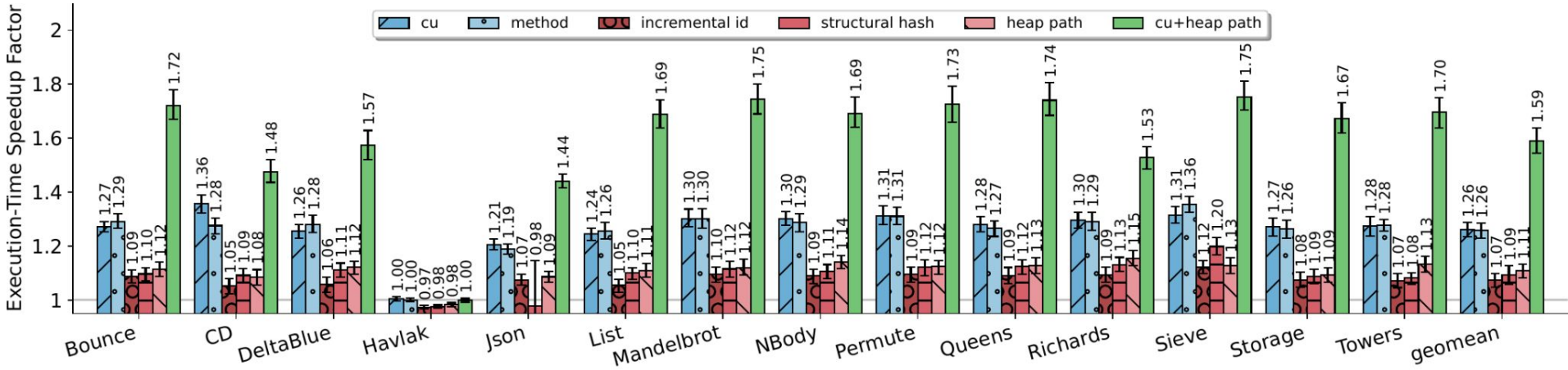
Evaluation - Performance (1)

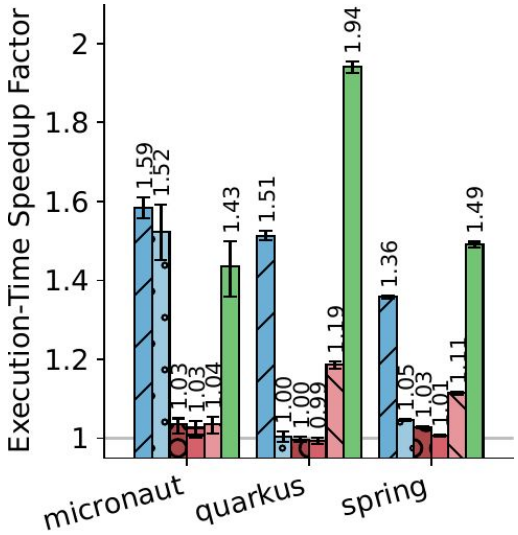




- Ordering strategies are effective
- Code ordering strategies lead to speedups up to 1.59×
  - On average, 1.26× (AWFY) and 1.48× (microservices)
- Heap ordering strategies lead to speedups up to 1.20×
  - On average, 1.11× (AWFY and microservices)
- Combined average speedup of **1.59×** (AWFY) and **1.61×** (microservices)

Evaluation - Performance (2)





- Code and heap orderings are synergistic
  - Code ordering affects the content of data structures storing metadata
  - We are investigating memory accesses causing blocking I/O

Conclusions

- We proposed a profile-guided methodology to improve the startup time of Native Image binaries by reordering the code and the heap-snapshot sections of the binary, reducing I/O traffic
- We described two code-ordering strategies and three heap-ordering strategies
  - Divergences in inlining decisions between builds of the same program
  - Matching objects from a profile against the objects in the profile-guided build
- We implemented our methodology and ordering strategies in GraalVM Native Image
- We evaluated our implementation on the “Are We Fast Yet?” benchmark suite and on widely-used microservice frameworks
  - Effective in reducing page faults and improving runtime performance

Thanks for your attention

- Artifact
  - DOI: https://doi.org/10.5281/zenodo.13302630
  - Docker image: https://doi.org/10.5281/zenodo.13302630
- Contacts:

Matteo Basso

matteo.basso@usi.ch