Initialize Once, Start Fast: Application Initialization at Build Time

CHRISTIAN WIMMER, Oracle Labs, USA
CODRUT STANCU, Oracle Labs, USA
PETER HOFER, Oracle Labs, Austria
VOJIN JOVANOVIC, Oracle Labs, Switzerland
PAUL WÖGERER, Oracle Labs, Austria
PETER B. KESSLER, Oracle Labs, USA
OLEG PLISS, Oracle Labs, USA
THOMAS WÜRTHINGER, Oracle Labs, Switzerland

Arbitrary program extension at run time in language-based VMs, e.g., Java's dynamic class loading, comes at a startup cost: high memory footprint and slow warmup. Cloud computing amplifies the startup overhead. Microservices and serverless cloud functions lead to small, self-contained applications that are started often. Slow startup and high memory footprint directly affect the cloud hosting costs, and slow startup can also break service-level agreements. Many applications are limited to a prescribed set of pre-tested classes, i.e., use a closed-world assumption at deployment time. For such Java applications, GraalVM Native Image offers fast startup and stable performance.

GraalVM Native Image uses a novel iterative application of points-to analysis and heap snapshotting, followed by ahead-of-time compilation with an optimizing compiler. Initialization code can run at build time, i.e., executables can be tailored to a particular application configuration. Execution at run time starts with a pre-populated heap, leveraging copy-on-write memory sharing. We show that this approach improves the startup performance by up to two orders of magnitude compared to the Java HotSpot VM, while preserving peak performance. This allows Java applications to have a better startup performance than Go applications and the V8 JavaScript VM.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Additional Key Words and Phrases: compiler; ahead-of-time compilation; virtual machine; optimization; Java; Graal; GraalVM

ACM Reference Format:
Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (October 2019), 29 pages. https://doi.org/10.1145/3360610

Authors' addresses: Christian Wimmer, Oracle Labs, USA, christian.wimmer@oracle.com; Codrut Stancu, Oracle Labs, USA, codrut.stancu@oracle.com; Peter Hofer, Oracle Labs, Austria, peter.hofer@oracle.com; Vojin Jovanovic, Oracle Labs, Switzerland, vojin.jovanovic@oracle.com; Paul Wögerer, Oracle Labs, Austria, paul.woegerer@oracle.com; Peter B. Kessler, Oracle Labs, USA, pkessler@amperecomputing.com; Oleg Pliss, Oracle Labs, USA, oleg.pliss@gmail.com; Thomas Würthinger, Oracle Labs, Switzerland, thomas.wuerthinger@oracle.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART184
https://doi.org/10.1145/3360610

1 INTRODUCTION

The recent shift to cloud computing and microservices has changed how server applications are written and deployed: Instead of using large application servers, independent services are deployed individually and executed on demand. When the load on a particular service increases, the underlying platform (e.g., AWS Lambda [Amazon Web Services, Inc 2019], IBM Cloud Functions [IBM 2019], Microsoft Azure Functions [Microsoft Azure 2019], Google Cloud Functions [Google Cloud Platform 2019], or Oracle Functions [Smith 2018]) spawns new workers to process the incoming workload. The first request processed on the newly spawned worker (*cold start*) requires initialization of the language runtime and initialization of the service configuration before the actual request is processed.

Cold start of VMs can be unpredictable [Barrett et al. 2017] and an order of magnitude slower than subsequent executions [Akkus et al. 2018], which can break the service-level agreement of a cloud service. Some language platforms, such as Java VMs and .NET, are particularly slow at startup because they perform expensive code verification, class loading, bytecode interpretation, profiling, and dynamic compilation.

In this paper, we explore a novel approach for reducing the startup time and memory footprint: combining points-to analysis, application initialization at build time, heap snapshotting, and ahead-of-time (AOT) compilation. This preserves the benefits of the Java ecosystem: combining existing libraries that are only loosely coupled and extend each other using, e.g., interfaces and declarative configuration files. Our approach is based on a closed-world assumption, i.e., all Java classes must be known and available at build time. At build time, the Java application and all its libraries (including the JDK) are processed by the points-to analysis to find the reachable program elements (classes, methods, and fields), so that only the necessary methods of Java classes are compiled.

Initialization code of the application can run at build time instead of at run time. The application can control what is initialized and allocate Java objects to build complex data structures. These objects are available at run time using a so-called *image heap*, a pre-initialized part of the heap that is part of the executable and available immediately at application startup. The points-to analysis can make objects reachable in the image heap, and the snapshotting that builds the image heap can make new methods reachable for the points-to analysis. These steps are executed iteratively until a fixed point is reached.

With our approach, only few initialization steps are executed at run time before the application's main method is invoked, for example, memory-mapping of the image heap. Multiple processes that run the same executable, or multiple isolated VM instances within one process, use copy-on-write sharing of the image heap. This reduces the overall memory footprint when multiple instances of the same application are started.

We combine ideas from several areas of prior research: The concept of an image heap has been used before by metacircular VMs [Alpern et al. 1999, 2005; Rogers and Grove 2009; Wimmer et al. 2013]. Heap snapshotting has been used by Smalltalk [Ingalls 1983] and Self [Ungar 1995], where development and deployment were not distinguishable and both code and data were kept in snapshots. Points-to analysis [Hind 2001; Ryder 2003; Smaragdakis and Balatsouras 2015; Sridharan et al. 2013] and AOT compilation are frequently applied and well-studied. The contribution of this paper is the combination of these ideas: an iterative application of points-to analysis and heap snapshotting.

We believe that our approach is more suitable for microservices than checkpoint/restore systems, e.g., CRIU [CRIU 2019], that restore a Java VM such as the Java HotSpot VM: Restoring the Java HotSpot VM from a checkpoint does not reduce the memory footprint that is systemic due to the dynamic class loading and dynamic optimization approach, i.e., the memory that the Java HotSpot

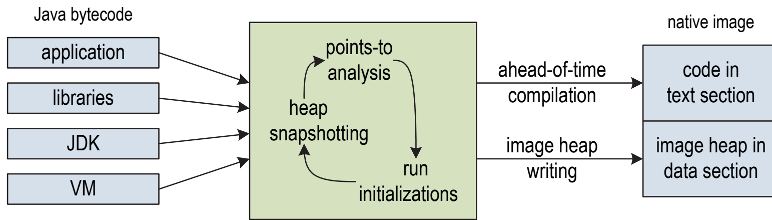

Fig. 1. Components executed at image build time to build a native image.

VM needs for class metadata, Java bytecode, and dynamically compiled code. In addition, it cannot rely on a points-to analysis to prune unnecessary parts of the application.

Our implementation, the Native Image component of GraalVM [Oracle 2019a], is written in Java and Java is used for all examples in this paper, but our approach is not limited to Java or languages that compile to Java bytecode. It can be applied to all managed languages that are amenable to points-to analysis, such as C# or other languages of the .NET framework.

In summary, this paper contributes the following:

- We run parts of an application at build time and snapshot the objects allocated by this initialization code, using an iterative approach that is intertwined with points-to analysis.
- We use points-to analysis results to only AOT-compile the parts of an application that are reachable at run time.
- We allow multiple processes, and multiple isolated VM instances within one process, to share the image heap using copy-on-write memory mapping.
- We measure the startup behavior and peak performance, and show that copy-on-write sharing of the image heap has significant benefits.

2 SYSTEM OVERVIEW

This section presents the system structure and details for all of the build-time and run-time components of GraalVM Native Image. The input of our system is Java bytecode, compiled from any language that compiles to Java bytecode such as Java, Scala, or Kotlin. The application, its libraries, the JDK, and VM components are all processed the same way. The result is a native executable for a specific operating system and architecture. We call such an executable a *native image*. Figure 1 shows the overall build-time system structure. First, points-to analysis and heap snapshotting are executed iteratively until a fixed point is reached. Callbacks registered by the application are also executed as part of this iterative approach, i.e., the application can participate. The result of this stage is a list of reachable classes, methods, and fields; as well as an object graph of reachable objects. Then, the reachable methods are compiled to machine code, and the object graph is written out as the image heap in the same layout that is used at run time for the heap. The machine code is stored in the text section of the native image, and the image heap is stored in the data section.

We call the entire process that produces a native image the *image build* time, to clearly distinguish it from the compilation of Java source code to bytecode (often called “compile time” or “build time”). The *image builder* is a Java application. The same Java VM that executes the points-to analysis and AOT compilation also runs the class initializers and the initialization callbacks of the application. This means that at image build time, objects that later form the image heap are ordinary objects in the Java heap of the image builder. Heap snapshotting discovers the objects that are reachable for the image heap.

One of the benefits of our approach is the blurred line between build time and run time: It is easy to move initializations between build time and run time without large code changes, i.e., it is possible to write an application that can support both configuration at build time and configuration at run time. Section 4 presents a concrete case study of such an application.

2.1 Points-to Analysis

We use a points-to analysis [Smaragdakis and Balatsouras 2015; Sridharan et al. 2013] to determine which classes, methods, and fields are reachable at run time. The points-to analysis starts with all entry points, e.g., the main method of the application, and iteratively processes all transitively reachable methods until a fixed point is reached. Our analysis is context insensitive, path sensitiv, and flow insensitive for fields but flow sensitive for local variables because it is based on SSA form [Hind 2001; Ryder 2003]. This subsection provides some background on our points-to analysis implementation. Note that this paper does not contribute a novel points-to analysis approach.

The points-to analysis uses the front end of our compiler to parse Java bytecode into the compiler's high-level intermediate representation (IR). The IR is then converted to a so-called *type-flow graph*. Nodes in the graph are instructions that operate on object types. Edges are directed *use* edges between nodes, i.e., they point from the definition towards the usage. Each node maintains a *type state*: a list of types that can reach this node and nullness-information. Type states are propagated through the use edges: if the type state of a node is changed, the change is propagated to all usages. Type states can only grow, i.e., new types can be added to a type state but types are never removed. This ensures that the analysis eventually reaches a fixed point and terminates.

The type-flow graph is a single interprocedural graph that covers the whole application. For `static` method calls, the caller and callee are linked when the type-flow graph is created: A use edge connects an actual argument node with the matching formal argument node, i.e., types flow into methods using argument nodes. The return node of the method has a use edge to the invocation node in the caller, i.e., types flow out of methods using return nodes. For virtual and interface method calls, the caller and callees are linked dynamically while the analysis is running: the type state of the receiver determines which callees are reachable. When a new type is added to the type state of the receiver, the new type is used to resolve the invocation. This ensures that types are correctly propagated through virtual method calls, but only necessary callees are linked. For each type, the points-to analysis tracks if the type is instantiated. Allocation bytecodes mark a type as instantiated, and are one of the sources of types in the type-flow graph (type-flow nodes without a predecessor). For each field, the points-to analysis tracks if the field is read or written. Reads and writes are tracked separately to find fields that are only read but never written at run time. Such fields are constant-folded later during AOT compilation.

2.2 Run Initialization Code

When the points-to analysis reaches a local fixed point (no more types added to type states), initialization code is executed. It comes from two different sources:

First, class initializers are executed. In Java, every class can have a class initializer (sometimes also called "static initializer") that is represented as a method named `<clinit>` in the class file. It computes the initial value of `static` fields. We allow the developer to decide which classes are initialized at image build time, and which classes remain uninitialized at image build time and are then initialized at run time. Section 3.1 provides more information on this policy choice.

Second, the application can register explicit callbacks that are invoked at build time. The application can run custom code, e.g., before, during, and after the analysis stage, by implementing hooks provided by our `Feature` interface. Figure 2 shows a fragment of this API.

```
interface Feature {
  // Hooks that are invoked at various stages at image build time.
  void beforeAnalysis(BeforeAnalysisAccess access);
  void duringAnalysis(DuringAnalysisAccess access);
  void afterAnalysis(AfterAnalysisAccess access);
}

interface DuringAnalysisAccess {
  // Access to the current points-to analysis state.
  boolean isReachable(Class clazz);
  boolean isReachable(Field field);
  boolean isReachable(Executable method);
}
```

Fig. 2. Fragment of the API to run application code at build time.

The relevant hook to run code before heap snapshotting is the "during analysis" hook: it is executed each time the points-to analysis has reached a local fixed point, but before the corresponding heap snapshotting. At this time, the application can run custom code that, e.g., allocates objects and initializes larger data structures. The important point to notice is that the initialization code can access the current points-to analysis state: it can query if a type, method, or field was already marked as reachable by the points-to analysis using the various `isReachable()` methods provided by `DuringAnalysisAccess`. The application can use this information to build data structures that are optimized for exactly the reachable parts of the application. Section 2.8 and the case study in Section 4 present examples for running initializations at build time. These examples use the API shown in Figure 2. The complete API is documented on the GraalVM homepage [Oracle 2019a].

2.3 Heap Snapshotting

Heap snapshotting builds an object graph, i.e., the transitive closure of reachable objects starting with root pointers such as `static` fields. This object graph is written into the native image as the image heap, i.e., the initial heap when the native image is started. Figure 3 shows the core algorithm. The input of the algorithm is the state of the points-to analysis, `pointsToState`.

The root pointers of the image heap are `static` object fields that are marked by the points-to analysis as read, as well as values that were constant folded into methods such as our image singletons (see Section 3.5). These root values are first added to a worklist, and then the worklist is processed until it is empty. Every object is only once added to the worklist and processed. For this, a set with all reachable objects is maintained.

To build the transitive closure, object fields are followed by reading field values using reflection (remember that the image builder is a Java application). Only instance fields that are marked as read by the points-to analysis are considered, i.e., if a class has two instance fields but one of them is not marked as read by the points-to analysis, the object reachable from that field is not part of the image heap.

If the class of the fields's value was not yet seen as a possible type of the field by the points-to analysis, the class is registered as a field type. In such a case, the next run of the points-to analysis propagates this new type to all reads of the field, and to all transitive usages in the type-flow graph.

Object arrays are handled in a similar way. In contrast to fields, the points-to analysis only maintains a single set of types for object arrays (the Java type system does not properly distinguish between arrays of different types). Types added to this set are propagated by the points-to analysis through the type-flow graph.

```
List<Object> worklist = new ArrayList<>();
Set<Object> reachableObjects = new IdentityHashSet<>();

Set<Object> buildHeapSnapshot(PointsToState pointsToState)
  for (Field f : pointsToState.reachableStaticObjectFields()) {
     // Root object of the image heap: reachable static object field.
     add(f.readValue());
  }
  for (Method m : pointsToState.reachableMethods()) {
    for (Object c : m.embeddedConstants()) {
      // Root object of the image heap: constant from, e.g., an image singleton.
      add(c);
    }
  }

  while (!worklist.isEmpty()) {
    Object cur = worklist.pop();

    if (cur instanceof Object[]) {
      for (Object value : (Object[]) cur) {
        add(value);

        pointsToState.getObjectArrayTypes().addIfAbsent(value.getClass());
      }

    } else {
      for (Field f : pointsToState.reachableInstanceObjectFields(cur.getClass()) {
        Object value = f.read(cur);
        add(value);

        // Ensure type is in the type state of the field. The points-to analysis
        // later propagates added types through the type-flow graph.
        pointsToState.getFieldValueTypes(f).addIfAbsent(value.getClass());
      }
    }
  }

  return reachableObjects;
}

void add(Object o) {
  if (!reachableObjects.contains(o)) {
    reachableObjects.add(o);
    worklist.push(o);
  }
}
```

Fig. 3. Algorithm for heap snapshotting.

The points-to analysis is executed again after heap snapshotting to propagate the updated type information through the type-flow graph. It reaches a new local fixed point, which triggers the run of new initializations. Then the heap snapshotting algorithm is run again. It starts with an empty set of reachable objects. It is not possible to start with the set of reachable objects produced by the previous run and just add new objects: The run of the points-to analysis in between could have marked new instance fields as used, i.e., `usedInstanceObjectFields()` can return a larger set of fields for any type, compared to the previous run of the heap snapshotting algorithm. Tracking changes and only processing the necessary objects that have changes would be more expensive than running the whole heap snapshotting algorithm again.

The global fixed point is reached when a points-to analysis run does not find any change in comparison to the previous run, i.e., when initializations and heap snapshotting did not make any new code reachable. The set of reachable objects returned by the last invocation of heap snapshotting represents the image heap: These objects are serialized and written into the data section of the native image.

Note that heap snapshotting does not make the points-to analysis context sensitive, e.g., heap sensitive or object sensitive. Adding context sensitivity to the points-to analysis is orthogonal to heap snapshotting.

2.4 Ahead-of-Time (AOT) Compilation

Only methods that are marked as reachable by the points-to analysis are AOT compiled to machine code and placed in the text section of the executable. We use the GraalVM compiler, an established compiler that compiles Java bytecode to machine code. The compiler is modular and can be configured for different VMs and compilation scenarios such as dynamic compilation and AOT compilation. However, the approach proposed in this paper is mostly independent from the compiler, and adapting our compiler to use points-to analysis results for AOT compilation instead of profiling information for dynamic compilation was not a significant amount of work.

The GraalVM compiler performs all standard optimizations such as method inlining, constant folding and arithmetic optimizations, loop optimizations; and optimizations that are especially effective for object-oriented Java code such as partial escape analysis [Stadler et al. 2014]. Speculative compiler optimizations that require deoptimization are disabled for the AOT compilation. Instead, the compiler incorporates points-to analysis results to improve the code quality: Fields that are not marked as written at run time are constant folded, regardless whether they are declared as `final` or not. Virtual and interface calls for which the points-to analysis found only a single callee are de-virtualized to a direct call, which also enables inlining of the callee. Type checks where the type state of the checked value only contains types that extend the checked type (or only types that do not extend the checked type) are replaced with a constant result. Similarly, null checks are removed when the points-to analysis marks a type state as never null.

Code that runs at image build time is often not reachable at run time and therefore not compiled. For example, class initializers that are executed at image build time are not compiled (see Section 3.1). However, it is allowed to have code that runs both at image build time and at run time, e.g., shared utility methods.

2.5 Image Heap

Execution at run time starts with an already pre-populated Java heap that is constructed by the heap snapshotting algorithm during image building: the *image heap*. Starting up the executable, or creating a new isolated VM instance (called *isolate*; see Section 2.6) in an existing process, must be fast and have a low memory overhead, so that many VM instances can be created for short-running tasks. Therefore, there must not be an expensive relocation step when loading the image heap. For isolates, it must be possible to map the image heap at different memory addresses in the same address space.

To achieve these goals, we use references that are relative to the start of the image heap. This means that memory accesses, like loading a field from an object or an array element, need more address arithmetic than the usual absolute references: the heap start must be added to the reference before the memory access. Objects of the image heap and objects allocated at run time use the same format, i.e., also objects allocated at run time use relative references. To make this as fast as possible, the heap start is always available in a fixed register (we use the register `r14` on x64 architectures).

Note that in many cases the addition can be folded into the memory access instruction on the x64 architecture, avoiding an explicit arithmetic operation.

With all object references being relative to the start of the image heap, the image heap that is prepared during image building and is part of the native image can be memory-mapped multiple times in the address space. This allows replicating the image heap without copying (fast isolate creation) and copy-on-write sharing of the image heap (low memory overhead).

The image heap is immortal, i.e., the garbage collector (GC) treats the image heap as root pointers. However, the GC only needs to scan a small part of the image heap for root pointers: Objects that are identified as read-only by the points-to analysis can never have references to objects allocated at run time and therefore contain no root pointers. Similarly, objects that contain only primitive data, including but not limited to arrays of primitive types, cannot contain root pointers. The image builder segregates objects in the image heap so that the GC only needs to scan a small, contiguous part of the image heap. Apart from this grouping of objects, we currently do not perform any layout optimization of the image heap.

2.6 Isolates

Isolates provide multiple independent VM instances in the same process. Instead of a single large heap, the developer can split up the heap into smaller pieces. Creating an isolate creates a new heap, with the image heap as the starting point. This means that all the initialization that is done during image building is immediately available in every isolate. All isolates share the same AOT compiled code, i.e., there is no separate points-to analysis and separate compilation per isolate. Since code is immutable, the sharing of code is desirable

Because each isolate has a separate heap, it is not possible to have direct Java object references between two isolates. This is a restriction and a benefit at the same time: The developer needs to ensure that the object graph is completely partitioned, and it is, for example, not possible to have a global cache that is accessible from all isolates. But the isolation allows garbage collection independently in each isolate, without stopping or influencing other isolates. All memory allocated by an isolate is freed automatically when the isolate is torn down, without the need for a garbage collection.

Note that we neither advocate to use multiple isolates within one process nor to use multiple processes with a single isolate each. It is up to the application developers to decide which model is best for them. This flexibility is not offered by most Java VMs such as the Java HotSpot VM, which only allows one VM instance per process.

2.7 Runtime Components

The native image contains a runtime system for Java, for example, a garbage collector (GC); stack walking and exception handling; and support for threads and synchronization. All of our runtime system is written in Java, discovered as reachable by the points-to analysis, and AOT compiled. There is no distinction between VM components, the application, libraries used by the application, and the standard Java library. As a result, the VM components also profit from AOT optimizations based on points-to analysis results, and VM components can, e.g., be inlined into application code. We use established techniques [Frampton et al. 2009; Wimmer et al. 2013] for writing low-level VM code in Java.

2.8 Example: Service Loaders

The closed-world assumption of our approach requires that all Java classes, i.e., the application and all its libraries, are known at image build time. However, this restriction must not impact how developers structure their application and must not prevent modular programming. For example,

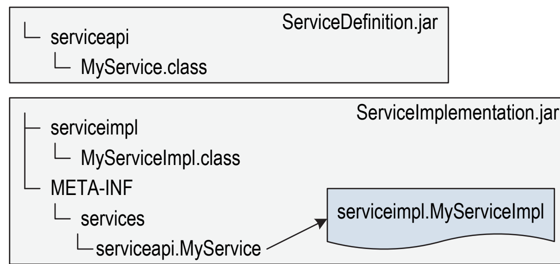

Fig. 4. Structure of classes and files expected by the Java `ServiceLoader`.

Java provides a standardized mechanism to load implementation classes of a specified service interface by declaratively specifying the class name of the service implementation classes. The implementation class names must be in a text file in the directory `META-INF/services`, and the file name must be the name of the service interface. The service interface and the implementations are usually in different .jar files, and likely implemented by different teams or even different companies.

Figure 4 shows how the service implementation class `serviceimpl.MyServiceImpl` that implements the service interface `serviceapi.MyService` is registered. A text file in the directory `META-INF/services` named `serviceapi.MyService` contains one line, specifying the class name `serviceimpl.MyServiceImpl`. The service implementation is loaded by the JDK class `ServiceLoader`: calling the method `load(MyService.class)` returns a newly allocated instance of the class `MyServiceImpl`.

Supporting service loaders in our system requires the iterative execution of points-to analysis and heap snapshotting: Always including all registered service interfaces and implementation classes in the native image would lead to an excessive amount of unnecessary code in the executable; but not supporting service loading would remove support for a large class of real-world applications. Because of the importance of service loaders, the approach described below is part of our standard distribution. However, it could be implemented by a developer too, i.e., it is not a deeply integrated part of our analysis but instead implemented as a `Feature`. Figure 5 shows a simplified version of the algorithm.

Before the first run of the points-to analysis, nothing is done for service interfaces or service implementation classes. Assume that the points-to analysis marks the service interface `MyService` of our example as being used. After that first round of the points-to analysis, only the service interface but not the service implementation class is marked as used. Now the `Feature` code to support service loaders runs. It iterates all service interfaces, and checks each interface whether it is marked as reachable via the API method `isReachable()`. This method returns `true` for the interface `MyService`. Therefore the file named `MyService` in the `META-INF/services` directory is loaded to retrieve the class names for the service implementations. In our example, this file contains one line: `serviceimpl.MyServiceImpl`. This is the name of the service implementation class. Using our reflection support (see Section 3.2), the class is registered for reflective lookup (via the API method `RuntimeReflection.register()`) and the `Constructor` object for the parameterless constructor of the class is created and made available at run time (via the API method `RuntimeReflection.registerForReflectiveInstantiation()`).

After the `Feature.duringAnalysis()` method, heap snapshotting is executed. The heap snapshotting sees the new `Constructor` object, which makes a new `Constructor.newInstance()` method reachable. That method contains an allocation bytecode for the service implementation class. The next run of the points-to analysis processes the new method, builds the type-flow graph,

```
class ServiceLoaderFeature implements Feature {

  Set<Class> processedServiceInterfaces = new HashSet<>();

  void duringAnalysis(DuringAnalysisAccess access) {
    // Iterate all files that contain service loader definitions.
    for (File serviceInterfaceFile : findMetaInfServicesFiles()) {
      Class serviceInterface = Class.forName(serviceInterfaceFile.getName());

      // Check if the service interface is reachable.
      if (access.isReachable(serviceInterface)) {

        // Process each interface only once (duringAnalysis runs multiple times).
        if (!processedServiceInterfaces.contains(serviceInterface)) {
          processedServiceInterfaces.add(serviceInterface);

          for (String serviceImplName : readAllLines(serviceInterfaceFile)) {
            Class serviceImplClass = Class.forName(serviceImplName);

            // Allow Class.forName at run time for the service implementation.
            RuntimeReflection.register(serviceImplClass);
            // Allow reflective instantiation at run time.
            RuntimeReflection.registerForReflectiveInstantiation(serviceImplClass);
          }
        }
      }
    }
  }
}
```

Fig. 5. Example `Feature` to support the Java `ServiceLoader`.

and because of the allocation bytecode registers the class `MyServiceImpl` as reachable. The new class is propagated through the type-flow graph, i.e., the class is added to type states. Interface method calls to the service interface now have a new concrete receiver type in the type state of the receiver, which means that the implementation methods of the service implementation class are marked as reachable. The whole process is repeated until a fixed point is reached, i.e., until the points-to analysis does not mark any new service interfaces as used. This means that the `duringAnalysis()` method runs again, but in our example does not find any new interfaces as reachable.

2.9 Limitations

The main limitation of our approach is the closed-world assumption: all application code needs to be available at image build time. Java reflection (see Section 3.2) and other dynamic introspection capabilities of Java are supported using configuration files provided by the developer at image build time. We believe that the closed-world requirement still allows a large and important class of applications to run on our system, especially cloud applications and microservices where startup time and memory footprint matter. Section 6 evaluates several microservice frameworks, including a complete application.

Several other limitations of our system are only a result of our implementation decisions and can be lifted without compromising the benefits of our approach. Currently, heap snapshotting only processes Java objects and no native resources. That excludes C memory that was allocated, e.g., using `malloc` or using Java's direct byte buffer; threads that are started at image build time; and file descriptors for files and sockets opened at image build time. It would be possible to capture the usages of native resources and, e.g., re-open files at run time that were opened at image build time.

3 IMPLEMENTATION DETAILS

This section presents details that are not novel contributions of the paper, but provide important background information to understand how the system works in practice and to understand the evaluation.

3.1 Class Initialization

Java provides for per-class initialization code: Before a class is used the first time (before the class is instantiated, before a `static` method is called, or before a `static` field is accessed the first time), the Java VM must execute the class initializer. In the Java class file, the class initializer is a method with the known name `<clinit>`. In Java source code, the class initializer can be written explicitly as a `static { ... }` block, but most classes have only implicit class initialization code: values directly specified at the declaration site of `static` and `static final` fields.

Class initialization code often allocates objects or even large object graphs, and makes them available via `static` fields. For most classes, the class initialization code does not depend on values that change at run time. Therefore, it is desirable to execute class initializers by default at image build time. The objects allocated by the class initializer are part of the image heap and therefore available at run time without initialization cost. However, some class initializers depend on run-time state. For example, a class initializer could query the current username or the current working directory, which change between image build time and image run time.

We use a combination of automatic analysis and manual developer input to determine the class initialization time: Class initializers that can be proven to not depend on run-time state are always initialized at image build time. For the remaining classes, the developer can list classes that can be initialized at build time. All other classes are initialized at run time.

A class marked for initialization at run time is initialized according to the Java specification, i.e., immediately before the first usage. The points-to analysis treats such a class initializer like any other method that can be invoked. For example, it correctly marks `static` fields written by the class initializer as "written at run time", even if the field is declared as `final`.

3.2 Reflection

The Java reflection API provides class lookup by name, as well as access of the methods, constructors, and fields of a class. Reflection complicates points-to analysis because it is not apparent which classes, methods, and fields are accessed [Landman et al. 2017]. A conservative analysis that assumes everything is accessed using reflection would defeat the goals of points-to analysis: we cannot determine on a fine-grained level which classes, methods, and fields are used by an application if all of them are possibly accessed using reflection. We require the developer to specify at image build time which classes, methods, and fields should be visible to reflection. Only the listed elements are then processed by the points-to analysis. To help collecting reflection information, we provide a tool similar to [Bodden et al. 2011]: a tracing agent that records reflection usage of the application either at development or build time.

The Java Native Interface (JNI) is handled in a similar way: JNI allows C code to look up classes, methods, and fields by name and then invoke the methods or access the fields. The developer needs to specify at image build time which elements are visible to JNI. This manual approach allows us to support reflection and JNI.

3.3 Support for Java 8 Lambdas

Since Java 8, the Java programming language supports lambda expressions. To decouple the language feature from implementation details, the Java bytecode generated for a lambda expression only

```
class ImageSingletons {
  // Add a singleton to the registry at image build time.
  static <T> void add(Class<T> key, T value) { ... }
  // Lookup a singleton from the registry.
  static <T> T lookup(Class<T> key) { ... }
}
```

Fig. 6. API to register and query singleton objects.

provides a high-level specification of the capture, but no actual implementation details. This is achieved on the Java bytecode level by using the `invokedynamic` bytecode. Invokedynamic is a version of a virtual call that does not have a target method fixed in the bytecode. Instead, the target method is provided by a *bootstrap method* that is executed once when the invoke is reached the first time. The bootstrap method for a lambda expression creates a new class, with bytecode assembled by the bootstrap method using the *ASM* bytecode generation framework.

Our approach does not support generating new classes at run time, because the points-to analysis needs to process all classes. Therefore, we force execution of the bootstrap method before the points-to analysis. Since the bootstrap method is side-effect free apart from the newly produced class, this does not change the behavior of the Java application. In summary, Java 8 lambda expressions are fully supported by our approach.

Several other parts of Java that normally rely on bytecode generation are handled similarly. For example, we force eager bytecode generation for proxy classes required by `java.lang.reflect.Proxy`.

3.4 Profile-Guided Optimizations

Java VMs use feedback-directed optimizations to focus optimizations on the most important parts of an application. The lower execution tiers (the interpreter or code compiled by a fast dynamic compiler) collect profiling information that higher tiers (the optimizing compiler) use to specialize the dynamically compiled code to the observed application behavior. Deoptimization [Hölzle et al. 1992] can be used to discard optimized code and continue execution in the lower tiers, which allows the optimizing compiler to perform speculative optimizations.

AOT compilation, including our approach, does not have profiling information from the current application run available at compile time. Therefore, we fall back to traditional offline profiling that is also used by C/C++ compilers: a special build of the application contains profiling counters. This build is used to run representative workloads, and on exit produces a file with the profiling information. The subsequent optimized build uses the profiling information to optimize the application, e.g., inline frequently executed method calls to reduce the method call overhead. Note that profile-guided optimizations are usually more important for Java than for C/C++. Java code tends to have smaller methods, more frequent method calls, and especially more frequent virtual method calls because all regular calls of instance methods are virtual calls. We use the profile-driven method inlining algorithm of GraalVM [Prokopec et al. 2019] also for AOT compilation when offline profiling information is available.

3.5 Image Singletons

Java uses `static final` fields to represent constants. These fields are optimized as expected by our points-to analysis and AOT compiler only if the class is initialized at image build time (see Section 3.1). Therefore, we provide a dedicated key-value store for compile-time constants that we call *image singletons*. Objects are registered in the store at build time (and only at build time) and are part of the image heap at run time.

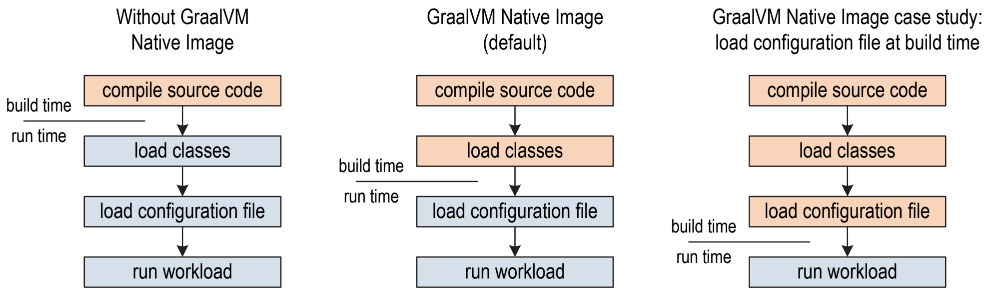

Fig. 7. Simplified application lifecycle.

Figure 6 shows the API definition. The key-value store lookup is constant folded before the points-to analysis, i.e., the object registered in the store is guaranteed to be a compile-time constant already when the points-to analysis runs (and later on during AOT compilation too). Section 4 provides an example usage of this mechanism.

4 CASE STUDY: LOAD A CONFIGURATION FILE AT BUILD TIME

Figure 7 shows the simplified lifecycle of a Java server application. In the traditional Java execution model, source code is compiled to class files at build time but everything else happens at run time. Classes are loaded and initialized, then a configuration file is loaded that determines, e.g., which services are available, and finally the workload is executed. Note that in reality the flow is not that linear, because classes are loaded and initialized on demand. In a large deployment where services are replicated on many servers, the configuration file does not change often. In fact, it might be that the configuration changes only when the application classes themselves are updated. In such a scenario, the startup step is replicated too, which consumes unnecessary resources and in a cloud deployment increases costs.

With our approach, by default classes are loaded at image build time. But since the application can participate in the initialization and heap snapshotting, it is also easy to load a configuration file at image build time and start with an already pre-configured application at run time. The native image is then tailored to a particular configuration. Since the points-to analysis iteratively discovers reachable application parts only after the configuration file is loaded, services that are not activated in the configuration file are also not marked as reachable and not included in the native image. The AOT compiler can, e.g., devirtualize calls and perform constant folding based on the points-to analysis results.

Figure 8 shows a minimal but nevertheless fully functional example of an application configuration. A configuration file in the commonly used JSON format specifies a configuration object, which in our case only provides the class name of a service handler implementation class. The service handler returns a `String` that is then printed by the application. Two service handler implementation classes are available: one always returns "Hello, world!", the other one returns the current date and time (see Figure 8a). We use the popular *Jackson* framework [FasterXML 2019] to parse the JSON file. The framework handles all of the parsing and allocates the `Config` object (see Figure 8b). Frameworks like Jackson are popular because they infer the information needed for parsing large object graphs automatically from fields, methods, and constructors of Java classes, and the application can influence the automatic behavior using annotations. In our case, it is enough to annotate the constructor parameter with `@JsonProperty` to define the JSON property `handlerClassName`. The JSON file shown in Figure 8c selects the `HelloWorldHandler`.

```
interface Handler {
  String handle();
}

class HelloWorldHandler implements Handler {
  String handle() { return "Hello, world!"; }
}

class CurrentTimeHandler implements Handler {
  String handle() { return new Date().toString(); }
}
```

(a) Declaration of handler interface and implementation classes.

```
class Config {
  Handler handler;

  Config(@JsonProperty("handlerClassName") String className) {
    handler = (Handler) Class.forName(className).newInstance();
  }

  static Config loadFile() {
    String fileName = System.getProperty("configFileName");
    return new ObjectMapper().readValue(new File(fileName), Config.class);
  }
}
```

(b) Configuration class and invocation of Jackson parser to load JSON file.

```
{ "handlerClassName" : "HelloWorldHandler" }
```

(c) JSON file to select `HelloWorldHandler` implementation class.

```
class ConfigureAtRunTime {
  static void main(String[] args) {
    Config config = Config.loadFile();
    System.out.println(config.handler.handle());
  }
}
```

(d) Traditional approach: load configuration at run time.

```
class ConfigureAtBuildTimeFeature implements Feature {
  void beforeAnalysis(BeforeAnalysisAccess access) {
    // This code runs at image build time and is not reachable at run time.
    ImageSingletons.add(Config.class, Config.loadFile());
  }
}

class ConfigureAtBuildTime {
  static void main(String[] args) {
    Config config = ImageSingletons.lookup(Config.class);
    System.out.println(config.handler.handle());
  }
}
```

(e) Case study approach: load configuration at build time and use it at run time.

Fig. 8. Case study: A configuration file in the JSON format, providing the implementation class of the `Handler` interface, is loaded using the Jackson framework. Exception handling and `public` keywords are elided for readability.

The class `ConfigureAtRunTime` (see Figure 8d) represents the "load configuration" and "run workload" stages. When running this class on the Java HotSpot VM, the involved classes are loaded on demand, i.e., all the Jackson classes are loaded when the configuration file is parsed. When running the same class using our approach, classes are loaded at build time. Still, the configuration file parsing happens at run time because the parser is invoked in the `main()` method. Both services could be selected in the JSON file. Since Jackson and also our application use Java reflection to instantiate the `Config` object and the service implementations, our reflection support (see Section 3.2) must be used to register these classes. The code for this is not shown in the figure.

In contrast, the class `ConfigureAtBuildTime` (see Figure 8e) shows how the developer can load the configuration file at build time. It uses the API introduced in Figure 2 and Figure 6. An image singleton (see Section 3.5) is used to store the `Config` object and make it available at run time (a `static` field could be used too, but would be less explicit in the source code). The JSON file to select the `HelloWorldHandler` must now be specified at image build time. The points-to analysis no longer sees the Jackson parser as reachable. This reduces the size of the native image. Also, the AOT compiler can better optimize the application: The `Config` object is a compile-time constant, and the field `handler` loaded from the `Config` object is constant folded too because the points-to analysis does not see any write to it. Therefore, the AOT compiler devirtualizes the invocation of the interface method `Handler.handle()` to the concrete implementation class `HelloWorldHandler.handle()`. Since this is a small method, it can be inlined too. In summary, the example source code is optimized to the same machine code as a direct call of `System.out.println("Hello, world!")`. All the abstractions of the service configuration are optimized away. The example also shows that the fine-grained points-to analysis is important: even though the class `Config` and its field `handler` are used at run time, the method `loadFile()` and the constructor are not reachable from the `main()` method and therefore not compiled.

Loading the configuration at build time reduces the startup time and the memory footprint of the application. When loading the configuration at build time, the instruction count is reduced from 650,000 to 530,000 instructions. The maximum resident memory set size is reduced from 7.64 MByte to 3.31 MByte. Figure 9 in the evaluation section provides more details for the different configurations.

Note that we separate the classes `ConfigureAtRunTime` and `ConfigureAtBuildTime` only for clarity of the example. It would be possible to have one class that supports both build-time and run-time configuration file loading. The developer can then either provide a configuration file at build time (and specialize the application towards that particular configuration), or build a generic application that loads the configuration file at run time.

5 CASE STUDY: CONTEXT PRE-INITIALIZATION FOR A JAVASCRIPT ENGINE

In this case study, we show how our concept of the image heap improves the startup of the high-performance JavaScript engine that is part of GraalVM. Note that from the point of view of this paper, the JavaScript engine is an application like any other: it is code written in Java that is analyzed by the points-to analysis and AOT compiled into a native image. JavaScript objects are stored in data structures written in Java, using arrays, maps, and hidden classes for fast access. A JavaScript object is often represented by only a single Java object storing data directly, but sometimes by a large graph of Java objects. Dynamic compilation is done by partial evaluation of the interpreter [Würthinger et al. 2017]. The dynamic compiler, the feedback directed optimizations, and the deoptimization system [Wimmer et al. 2017] necessary to achieve high JavaScript peak performance are also all written in Java and considered to be part of the application.

The JavaScript global object contains lots of built-in functions: the whole JavaScript standard library is available from it, and since in JavaScript everything is dynamic the functions of the

standard library can be modified like any other JavaScript object by writing properties. So the JavaScript global object is actually a reasonably large object graph. Creating this object graph at every startup of the JavaScript engine leads to high time and memory overheads. Even though the global object, and everything reachable from it, is mutable, real-world JavaScript applications tend not to overwrite properties because changing built-in functions is considered bad software-engineering practice. So when starting multiple JavaScript engines on the same server, copy-on-write sharing of the global object is desirable. The image heap provides these benefits: The JavaScript global object graph is created at build time by a custom `Feature` callback into the JavaScript engine and then stored in the image heap. When the JavaScript engine starts up, the image heap is copy-on-write mapped so that the read-only parts are automatically shared between multiple processes, or multiple isolates started within one process.

Initializing the global object at build time reduces the startup of our JavaScript engine (running "Hello, world!" written in JavaScript) from 7.63 million instructions to 1.70 million instructions. The native image size of the JavaScript engine increases by about 420 KByte. This increase comes from the additional objects in the image heap. Figure 9 in the evaluation section provides more details for the different configurations.

6 EVALUATION

This section evaluates the startup performance (startup time and memory footprint) and peak performance of our system. We use the GraalVM Enterprise Edition 19.2 release for this evaluation, which is based on the JDK 8 release 8u221. For comparisons with the Java HotSpot VM in this section, we use both the JDK 8 release 8u221 (to have a comparison that uses the same class files of the Java standard library) and the latest release JDK 12.0.2 (to allow an evaluation of the AOT compilation provided by the Java HotSpot VM that is not yet available in JDK 8, see Section 6.4). All measurements were performed on a dual-socket Intel Xeon E5-2690 v4 system with 14 physical cores per socket running at 2.6 GHz, 768 GByte main memory, running Oracle Linux Server release 7.4. Benchmarks were executed on one of the CPUs, on one hardware thread per core, and with Turbo Boost disabled to avoid instability in the results due to non-uniform memory access, interference between hyper-threads, and boosted / non-boosted cores. All measurements were repeated 10 times, reported numbers are the average of the runs.

6.1 Startup Performance

To evaluate the startup performance and memory footprint of our approach, we evaluate "Hello, world!" in different languages and configurations. The 4 left columns of Figure 9 show the execution time (measured with the standard `perf` [Perf Wiki 2019] tool), executed user-space instructions (the `instructions:u` counter of `perf`), the total number of executed instructions (the `instructions` counter of `perf`), and the maximum resident memory set size. The memory size is reported by the operating system and therefore includes the Java (or JavaScript) heap, off-heap memory, stacks, and the parts of the executable loaded into memory.

The canonical "Hello, world!" in C executes in 1 millisecond and 90,000 user-space instructions. It has a memory footprint of 1.1 MByte. In contrast, the canonical "Hello, world!" in Java executed on the Java HotSpot VM of JDK 8 requires about 100 milliseconds, 152 million instructions, and 37.5 MByte. Some of these numbers improve slightly with JDK 12, some regress. However, there are no major differences between JDK 8 and JDK 12 for any of the numbers in Figure 9.

The native image built by our approach makes the startup of Java competitive with C again: It executes in 1.9 milliseconds, 530,000 instructions, and 3.37 MByte. Note that this includes the Unicode-to-UTF8 conversion done by `println()` since Java uses Unicode strings. The native image size of our binary is 1.87 MByte. The image heap accounts for 0.94 MByte of that. It includes all the

	Execution time [msec]	Instructions (user space) [millions]	Instructions (total) [millions]	Maximum memory size [MByte]	Native image size [MByte]	Image heap size [MByte]
"Hello, world!"						
C	0.9	0.09	0.43	1.10	0.02	
Go	1.9	0.47	1.23	7.29	1.92	
JDK 8, Java HotSpot VM	103.9	152.45	171.41	37.50		
JDK 12, Java HotSpot VM	74.8	119.13	144.09	52.26		
JDK 12, Java HotSpot VM with AOT	89.1	119.60	147.82	69.21	46.61	
GraalVM Native Image	1.9	0.53	1.98	3.37	1.87	0.94
Case study: load a configuration file						
JDK 8, Java HotSpot VM	438.8	1,281.12	1,325.75	65.07		
JDK 12, Java HotSpot VM	414.7	1,370.30	1,420.36	75.87		
JDK 12, Java HotSpot VM with AOT	224.6	332.99	377.63	123.13	125.44	
GraalVM Native Image (load at run time)	2.6	0.65	2.81	7.64	8.52	3.92
GraalVM Native Image (load at build time)	1.9	0.53	1.97	3.31	1.87	0.94
Case study: JavaScript "Hello, world!"						
V8 JavaScript VM	14.5	8.62	18.52	22.64	34.90	
GraalVM without context pre-initialization	8.1	7.63	12.98	21.53	90.23	55.24
GraalVM with context pre-initialization	5.0	1.70	6.46	19.70	90.65	55.66
Quarkus						
JDK 8, Java HotSpot VM	983.4	3,172.07	3,259.99	160.10		
JDK 12, Java HotSpot VM	930.2	4,086.90	4,197.35	145.61		
JDK 12, Java HotSpot VM with AOT	540.8	926.26	1,003.80	221.66	181.36	
GraalVM Native Image	10.5	5.36	14.60	16.30	16.62	7.82
Micronaut						
JDK 8, Java HotSpot VM	1,966.8	7,694.26	7,818.66	198.31		
JDK 12, Java HotSpot VM	2,011.9	8,370.03	8,549.05	169.49		
JDK 12, Java HotSpot VM with AOT	1,049.3	1,455.64	1,547.44	263.59	300.06	
GraalVM Native Image	30.0	38.31	53.18	37.50	45.82	19.09
Helidon						
JDK 8, Java HotSpot VM	978.9	3,424.57	3,500.54	106.98		
JDK 12, Java HotSpot VM	945.8	4,100.15	4,202.16	116.49		
JDK 12, Java HotSpot VM with AOT	525.5	855.29	925.80	191.79	187.54	
GraalVM Native Image	22.7	35.10	46.14	25.60	20.89	9.64

Fig. 9. Startup performance and memory footprint for "Hello, world!", the case studies, and Java microservice frameworks (left columns), and size of the native images (right columns).

necessary metadata for the GC and precise exception information (line number and file name for every bytecode that might throw an exception).

Our native image is comparable to, or performs better than, the canonical "Hello, world!" in Go (version go1.12.7). We are evaluating Go because it is a direct competitor to our approach: it is at the same point (managed, statically compiled) in the design space as our approach, i.e., in between C (unmanaged, statically compiled) and the JDK (managed, dynamically compiled).

The second group of rows in Figure 9 evaluates the case study presented in Section 4. The Jackson parser for the JSON file adds a significant startup overhead to the Java HotSpot VM: the execution

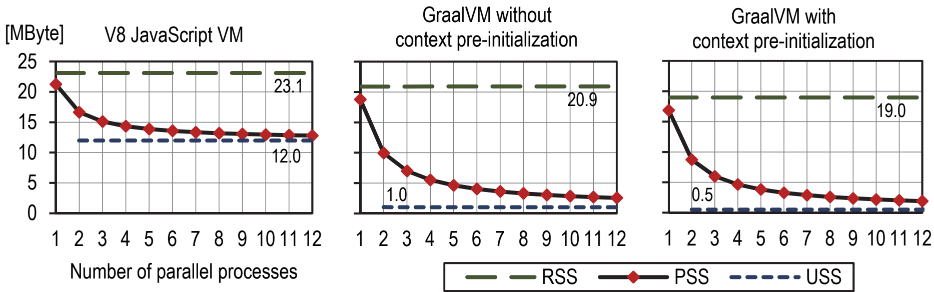

Fig. 10. Memory sharing when starting the JavaScript engines multiple times in parallel (lower is better).

time increases from 100 milliseconds for the canonical “Hello, world!” to more than 400 milliseconds to parse the JSON file into a `Config` object, load the class listed in the configuration, and print the “Hello, World!” string returned by the `HelloWorldHandler`. For our native image, the increase is not as dramatic but still significant when loading the configuration file at run time. For example, the memory footprint increases from 3.37 MByte to 7.64 MByte. When loading the configuration file at build time, all numbers go down to the same value as the canonical “Hello, world!”: the same machine code is created because all accesses of the `Config` object are constant folded.

The third group of rows in Figure 9 evaluates the JavaScript case study presented in Section 5. We use the state-of-the-art V8 JavaScript VM (version 7.6.303.29) as the baseline for a canonical “Hello, world!” written in JavaScript. V8 has an excellent startup behavior, and deserializes a pre-initialized JavaScript context from an external file on startup [Guo 2015]. Our JavaScript VM starts faster than V8: with the context pre-initialization presented in Section 5, the execution time is 5.0 milliseconds (1.70 million instructions). Without the context pre-initialization, the JavaScript global object needs to be built at run time. This increases startup time to 8.1 milliseconds (7.63 million instructions). The context pre-initialization has a small but measurable impact on the image heap: the size of the image heap increases by about 420 KByte when using context pre-initialization.

6.2 Copy-on-Write Memory Sharing

The maximum resident memory set size (RSS) reported in Figure 9 does not show the actual physical memory usage when the same process is started multiple times (or multiple isolates are created within one process) because RSS includes shared memory for every process, i.e., it reports shared memory multiple times. To evaluate the copy-on-write sharing of our image heap, we use the tool `smem` to also query the Unique Set Size (USS) and Proportional Set Size (PSS). The USS contains only the memory that is private to a process, not reporting any shared memory. The PSS attributes shared memory proportionally, i.e., for memory shared between N processes every process reports 1/N of the shared memory. The sum of all PSS is, therefore, the total physical memory usage, while the USS is a hint for how much more physical memory is necessary when another process is started.

Figure 10 shows the memory behavior when the JavaScript engines (V8 and our engine without and with context pre-initialization) are started multiple times in independent processes. We start the JavaScript engines and print “Hello, world!” using the interactive console of the engines, so that the engines remain running waiting for more input. Then we query the memory sizes with `smem`. The figure shows the results for 1 to 12 engines started in parallel. With only 1 process, the RSS size is about the same as the memory size reported in Figure 9 (the small differences are due to the console mode used in this experiment). The PSS for one process is close to the RSS because only system libraries can be shared with other processes. The USS for 1 process is the same as the

PSS because there is no other process to share, so we do not report it. Starting with 2 processes, the PSS goes down with every added process because the shared memory is shared between more processes. The lower limit is the USS: every new process requires this amount of new physical memory. All numbers are per-process, therefore the RSS and the USS do not change when more processes are started.

The V8 JavaScript VM shows only a modest amount of memory sharing between processes, most likely only the code (the text segment) of the executable. For our JavaScript engine, the image heap is also shared. Only a small part of the image heap is modified by the processes, therefore most of the image heap is shared due to the copy-on-write memory mapping. Without context pre-initialization, the JavaScript context needs to be allocated and initialized at startup, leading to a USS of about 1 MByte. With context pre-initialization, the context is part of the shared image heap, which reduces the USS to about 0.5 MByte. This means that a new instance of our JavaScript VM only requires 0.5 MByte of physical memory once another VM instance is already running, while the V8 JavaScript VM requires 12 MByte. In summary, this experiment shows that our copy-on-write sharing of the image heap is beneficial.

6.3 Java Microservice Frameworks

The recent shift away from heavyweight Java application servers (such as Tomcat, JBoss, or WebLogic) to lightweight microservices lead to the release of several new Java microservice frameworks. We evaluate the startup performance of three such frameworks: Quarkus [RedHat 2019] (version 0.18.0), Micronaut [Object Computing 2019] (version 1.1.3), and Helidon [Oracle 2019b] (version 1.0.1). We measure all three frameworks in a `"Hello, world!"` configuration that exposes one web service bound to one http port. A startup hook instructs the frameworks to shut down immediately after the web service is up and running. This allows us to measure startup time in a consistent way, but as a side effect also includes shutdown time. Note that our numbers do not allow a meaningful comparison of the three frameworks with each other, because different features (such as security and authentication features) might be turned on or off by default, and because the startup hooks might run at slightly different stages during startup. Only the comparison that we are interested in, i.e., comparing the startup time on the Java HotSpot VM with the startup time of a native image, is valid based on these numbers.

The bottom three groups of Figure 9 show the startup performance of the three microservice frameworks. Our approach leads to nearly two orders of magnitude improvement in startup time and instruction count, and a significant improvement in memory size. The native images of all three frameworks start in 30 milliseconds or less, while the startup on the Java HotSpot VM takes a second or more.

6.4 AOT Compilation Provided by the Java HotSpot VM

The Java HotSpot VM allows an application developer to compile some classes ahead of time and ship a shared library with the code; and it provides a heap snapshotting mechanism called *Application Class Data Sharing* (AppCDS) for the class metadata of loaded classes to reduce the class loading time in subsequent runs. These two mechanism are independent of each other and also do not profit from each other. The “JDK 12 with AOT” rows in Figure 9 show the results when both mechanism are enabled.

We use the following steps to build the AOT images for the Java HotSpot VM:

- We perform a profiling run of the application with the option `-XX:DumpLoadedClassList` to find the list of classes loaded by the application. This list includes application classes, library classes, and JDK classes.

	Reachable classes	Reachable methods
"Hello, world!"	569	3,398
Runtime system, e.g., GC	290	1,768
JDK	278	1,629
Case study: load a configuration file (load at run time)	1,761	12,667
Case study: load a configuration file (load at build time)	570	3,399
Our JavaScript VM	10,366	68,825
Quarkus	3,763	22,816
Micronaut	9,625	57,863
Helidon	5,191	30,564

Fig. 11. Number of classes methods found as reachable by the points-to analysis.

- We use the tool `jaotc` for AOT compilation of these classes. This produces a shared library with all methods of all the listed classes. The tool cannot operate on a per-method granularity, i.e., the shared library contains methods even if they are unreachable.
- We use the option `-Xshare:dump` during a second profiling run to build the AppCDS information.

The measured benchmarking run then uses the option `-XX:AOTLibrary` to load the AOT code and the options `-Xshare:on` and `-XX:SharedArchiveFile` to enable AppCDS. The combined size of both archives is reported in the column "native image size" of the respective rows in Figure 9.

For the simplest Java "Hello, world!", the AOT compilation of the Java HotSpot VM does not provide any benefits. For our case study and the Java microservice frameworks, it reduces the execution time to about half and reduces the instruction count by a factor of 4 or more. In contrast, our native image reduces both time and instruction count by several orders of magnitude. In addition, the size of the archives for the Java HotSpot VM (which must be shipped in addition to the class files and in addition to the Java HotSpot VM itself) is always larger than the size of our native images (which are self-contained and do not require any additional files).

The benefits of AOT compilation for the Java HotSpot VM are limited because it does not change the expensive startup sequence of the VM; it still requires class initialization to run for every class; and it does not allow application objects to be stored in the AppCDS archive. This experiment indicates that AOT compilation of Java code without a closed world assumption and without a points-to analysis cannot come close to the startup behavior of our approach.

6.5 Reachable Classes and Methods

Figure 11 shows the number of classes and methods that are found as reachable by the points-to analysis for the various native images. For the "Hello, world!" image, only one class / method is from the application. About half of the classes are from our runtime system, for example the GC. The other half is from the JDK.

For our case study, the Jackson parser for JSON files increases the numbers by a factor of 3 to 4 when the configuration file is loaded at run time. Loading the configuration file at build time eliminates the parser completely. The class and method count is the same as for "Hello, world!", plus one class / method for `HelloWorldHandler.handle()` as shown in Figure 7. Our JavaScript VM is the largest image that we evaluate. The total image build time varies between 10 seconds for the "Hello, world!" image and 2 minutes for our JavaScript VM.

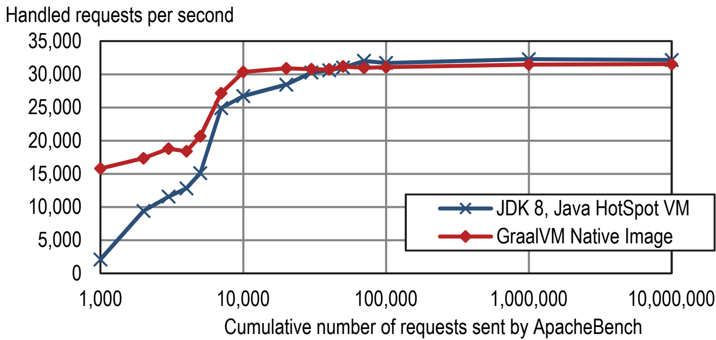

Fig. 12. Peak performance of a REST service based on the Quarkus microservice framework (higher is better).

6.6 Peak Performance

There are no established benchmarks yet for the Java microservice frameworks presented in the previous sections. Therefore, we create a simple benchmark that decodes and encodes JSON data. It exposes networking end-points to invoke the JSON functions. We benchmark the application using ApacheBench [The Apache Software Foundation 2019] to send http requests to a REST service endpoint. Each run starts with a cold VM. Measurements start after the service is running, i.e., the startup time of the VM reported in Figure 9 is excluded from the measurements.

Figure 12 compares the performance of our native image with JDK 8. The results for JDK 8 and JDK 12 are the same, therefore we omit JDK 12 from the figure for clarity. We use an exponentially increasing number of requests to the server, from 1,000 to 10,000,000 requests, and report the requests served per second as reported by ApacheBench. The number of requests are cumulative in the same VM instance: We first send 1,000 requests for the first data point, then 1,000 more requests to reach the 2,000 requests datapoint, and so on. This allows us to show both warmup and peak performance: For the low number of requests, the Java HotSpot VM has not reached its peak performance. It takes several seconds, up to 70,000 requests, until peak performance is reached. The 1,000,000 and 10,000,000 requests datapoints show the peak performance. Our native image reaches peak performance much faster. There is still a bit of a warmup curve because, e.g., memory needs to be requested from the operating system for the first requests. The peak performance is competitive with the Java HotSpot VM.

All configurations (native image, JDK 8, and JDK 12) use 32-bit compressed references. 32 bits are sufficient for references up to a maximum heap size of 32 GByte (2^{35} byte). The additional 3 bits of addressing range over a 32-bit reference are due to the 8-byte object alignment. In our system, it is straightforward to use only 32-bit offsets relative to the beginning of the heap instead of the full 64-bit offsets because all memory accesses are already relative to the beginning of the image heap. The native image is built with profile-guided optimizations (see Section 3.4).

6.7 Application Developer Effort

Our approach requires the application developer to provide certain input. For example, elements that are visible to reflection at run time (see Section 3.2) must be specified at image build time. A full usability study is out of scope for this paper. To provide some insight how much code is necessary, we analyze the integration code written by the Quarkus project [RedHat 2019]. Note that none of the paper authors is affiliated with the Quarkus project. Quarkus provides integration for a large number of open-source projects. Their integration code provides not only support for

	Size of project		Size of Quarkus integration	
	Files	Lines of code	Files	Lines of code
hibernate-orm	3,921	308,125	62	4,180
netty	1,841	196,838	5	373
undertow	875	102,707	43	3,219
kafka-client	700	62,421	6	336
vert.x	499	54,852	20	1,056
hibernate-validator	847	41,371	15	778
kafka-streams	416	36,812	3	82
jackson	115	26,565	1	12
flyway	290	17,107	10	419
jaeger-client-java	108	5,575	9	404

Fig. 13. Code size of projects that Quarkus provides native image integration for.

native image, but contains also parts that are used by their build system and when running on the Java HotSpot VM. While it is impossible to separate these parts, the integration code still provides some insight (and upper bound) on the developer effort.

Figure 13 lists some of the projects that Quarkus supports. The left two columns show the size of the project: the number of Java source code files and the lines of Java code, as reported by the tool `cloc`. The right two columns show the size of the Quarkus integration code. Test code is excluded from all numbers. There is no consistent ratio between the project size and the integration code size. But the size of the integration code is always orders of magnitude smaller than the project size.

Note that all integration code is provided by library and framework developers. None of the (arguably small) applications based on the microservice frameworks that we evaluate require any integration code.

For the case studies presented in this paper, the integration code is minimal. Loading a configuration file at build time as presented in Section 4 only requires the `Feature` implementation shown in Figure 8e. Our JavaScript VM was designed from the beginning to be compatible with the native image approach. For example, it does not use reflection at all in its implementation and therefore does not need a reflection or JNI configuration file. The code for context preinitialization, and native image support of our language implementations in general, is language independent. This support code is not well isolated and therefore difficult to measure. We identified about 2,000 lines of code that are specific for native image support.

7 RELATED WORK

The V8 JavaScript VM [Google 2012] uses heap snapshotting to pre-initialize the execution context [Guo 2015] like the global object and all the built-in functions. A snapshot is deserialized into the heap to avoid expensive computations every time a context needs to be created, but to the best of our knowledge no copy-on-write sharing is used. Snapshotting can be used by engine embedders to pre-initialize additional library scripts and further reduce startup time. However the V8 snapshotting mechanism does not discover and remove unreachable parts. The main limitation is similar to that of our system: the snapshot can only capture the V8 heap but not the native memory. Additionally, values derived from sources such as `Math.random` or `Date.now` are fixed once the snapshot has been captured. Our system avoids this problem due to the fine grained selection of class initializers that can run either ahead-of-time or at run-time. Similarly, the Dart VM [Google 2019a] uses snapshots [Google 2019b; Schuster 2011] to optimize application startup.

Process checkpoint/restore technology saves the current state of a whole process. This technique is widely used in container live migration. For example, *Checkpoint/Restore in Userspace* (CRIU) [CRIU 2019] for Linux relies on the *ptrace* kernel interface to seize the process. It performs incremental checkpoints creating a fork of the process being checkpointed and using page faults to detect memory writes as they occur. It has the advantage that it can capture most native resources such as open files. However, process-level checkpoints have an overhead when only a portion of an application needs to be checkpointed. When applied to the Java VM, a system like CRIU dumps the entire resident heap of the VM, including objects that are no longer reachable. To address this problem, ALMA [Bruno and Ferreira 2016] extends CRIU and the Java HotSpot VM with a migration-aware GC policy. It minimizes the amount of data being snapshotted by analyzing the heap to discover heap regions that should be collected before migration. Since this system targets container live migration over the network, it uses a cost-benefit estimation that compares the GC cost with the network bandwidth cost to determine which heap regions should be collected. It relies on the G1 GC [Detlefs et al. 2004] of the Java HotSpot VM, which periodically marks the heap and reports both an estimate of the amount of space that would be reclaimed if a particular region is collected, and of the time needed to collect a particular region. Alternatively, Crochet [Bell and Pina 2018] uses bytecode rewriting and automated proxy types to checkpoint and rollback individual objects in the Java VM. It performs a lazy heap traversal copying or restoring state as needed, starting from the heap root pointers. It instruments all field accesses to call a special method with specific behavior to copy the object and propagate the traversal. This technique enables fine-grained checkpoints at the cost of a steady state overhead, without requiring any modifications to the Java VM itself. While CRIU focuses on checkpointing for process migration, the goal in Crochet is to enable recovery within the same process as the checkpoint. In contrast to CRIU, the focus of our approach is to reduce the overhead of loading Java applications. Checkpoint/restore of a Java VM does not reduce the memory footprint that is systemic to the dynamic optimization approach of Java VMs. A restored VM still requires memory for class metadata, Java bytecode, and dynamically compiled code; and it cannot rely on a points-to analysis to prune unnecessary parts of the application. We believe that optimizations for process migration are orthogonal to our approach.

The transporter in the Self system [Ungar 1995] enables transfer of objects between *worlds*, dynamic ecosystems of live running objects. A live program can be serialized to a source file by using annotations on objects and slots (fields or methods) to control how information is saved. The live state of the program can then be recreated in another *world*. The Smalltalk-78 [Ingalls 1983] system introduced the idea of *cloned implementation* to enable transfer of the program state between machines and to improve portability. The Smalltalk-78 implementation was cloned from its predecessor, Smalltalk-76, using the *system tracer*, itself a Smalltalk program. The *system tracer* ran from within Smalltalk-76 and copied the entire system onto an image file. This file could then be loaded and executed by the Smalltalk-78 interpreter. The next versions of the Smalltalk system were similarly produced by repeating this process. Smalltalk also uses the snapshot concept for checkpointing: the developer saves the entire state upon program termination and resumes it a later time, possibly on a different machine. Automatically scheduled snapshots could be resumed to recover execution from fatal errors.

Lisp systems provide facilities to save the user's private memory to a file, i.e., a snapshot of a running Lisp, via *sysout* [Bobrow et al. 1969; Steele and Gabriel 1993; Teitelman 1974]. The state can be loaded later, potentially on a different machine, via *sysin*. It could also save the stack so that the subsequent *sysin* following a *sysout* could continue from that point. Although *sysout* would not save the state of open files one could specify what should be done when the saved image is restored, e.g., reconnect to servers and open files. An APL *workspace* [Falkoff and Iverson 1973;

Wheeler 1981] groups functions and variables intended to work together. The state of a *workspace* can be snapshotted to a file and restored as needed for inspection, amendment, and execution, while preserving the state of the computation during periods of inactivity. R [R Core Team 2019] uses a similar *workspace* [Chambers 2008] concept. In contrast to our work, the systems mentioned above do not use any points-to analysis, but rely solely on already executed code for the snapshot.

A widely used technique in VMs for resource-constrained mobile and embedded devices is *romization* [Aslam et al. 2010; Marquet et al. 2005; Titzer 2006; Titzer et al. 2007], where a VM is loaded from a pre-initialized state. Optimizations such as redundancy elimination [Pilipenko and Pliss 2018], i.e., elimination of unreachable methods, fields, and classes, can reduce the static and dynamic footprint of the VM. Analogously to our system, Virgil [Titzer 2006] explicitly separates initialization time from run time and allows applications to build complex data structures during compilation. However, Virgil makes an important simplifying assumption: the heap is fixed at compile time, there are no dynamic object allocations at run time. This is possible due to the problem domain for which the system was designed: resource-constrained embedded devices. A fixed heap allows for an efficient and precise analysis of the reachable code. The analysis is single pass (each method is processed at most once) and not iterative like our approach, so the initialization code cannot take points-to analysis results into account. The code size is significantly reduced and unused objects, meta-objects, and fields eliminated. This design constraint also obviates the need for runtime systems like a GC, allowing a Virgil program to run directly on the bare hardware without a VM or language runtime.

Optimization of a given application under closed-world assumptions is also known as application extraction. It reduces the code footprint of applications that rely heavily on libraries. Application extractors for Java can remove dead methods and fields, inline method calls, and simplify the class hierarchy. An example is the Jax system [Tip et al. 1995]. It relies on user input to specify the components that are accessed reflectively. Library subsets can be built on a per-application basis [Rayside and Kontogiannis 2002].

Agesen et al. [Agesen and Ungar 1994] describe an application extractor for the Self language based on type inference. Their technique is able to efficiently eliminate unused slots from objects. The extractor is fully automated with two exceptions: sends with computed selections and reflection.

Similarly to previous work our system performs application extraction by running the points-to analysis under closed-world assumptions and eliminating unreachable methods, fields, and classes, and employs romization techniques by pre-initializing classes at image build time.

Metacircular Java VMs such as the Jikes RVM [Alpern et al. 1999, 2005; Rogers and Grove 2009] and the Maxine VM [Wimmer et al. 2013] are Java VMs written in Java. They use a similar build time approach as our system that results in an image heap, but to the best of our knowledge, the image heap was never exposed to applications. The heap snapshots only contain VM objects, not application objects, and are not used to improve application startup. Our system also provides a runtime system for Java written in Java, however it is not a metacircular VM: we do not support loading new classes (or new code in general) at run time. In contrast to our approach, Jikes RVM and the Maxine VM have an open-world approach: the application is loaded and compiled at run time. They do not use a points-to analysis at build time because it is not possible anyway to specialize the AOT compiled code: code loaded at run time would invalidate the points-to analysis results. We believe that our approach is significantly different from metacircular VMs and cannot be retrofitted into metacircular VMs.

Partial evaluation of object-oriented languages such as Java has been extensively studied [Dean et al. 1994; Marquard and Steensgaard 1992; Schultz 2001; Schultz et al. 2003; Shali and Cook 2011]. Shali et al. [Shali and Cook 2011] use partial evaluation to specialize generic frameworks in the context of the usage pattern of particular applications. Schultz et al. [Schultz et al. 2003] present an

automatic program specializer for Java programs and show how partial evaluation can reduce the overhead of object-oriented abstractions in generic programs.

A related technique is compile-time function evaluation (CTFE), i.e., the ability of a compiler to evaluate a function at compile time. Through CTFE you can, for example, replace data stored as magic values in opaque data structures with an expressive computation that generates that data. CTFE is popular with major programming languages. In D and C++, CTFE is limited to side-effect free functions that operate on compile-time known values. In D [D Language Foundation 2019] the compiler may choose to evaluate some computations at compile-time depending on their complexity [Teoh 2017]. When the value of an expression must be known at compile-time, CTFE can be forced through keywords such as `static`, `immutable`, or `enum`. For non-trivial computations, the compilation can get excessively slow [Koch 2017]. In order to evaluate a function at compile time the compiler essentially has to compile its body into a state where it can be run inside a D interpreter embedded inside the compiler. At the time of this writing the CTFE implementation in the D language compiler relies on an AST interpreter. This means that for non-trivial computations the compile times can become extremely high and the compiler can exhaust the available RAM [Koch 2017]. There is, however, an ongoing effort to solve this issue by replacing the current CTFE engine with a bytecode interpreter based one that promises to offer superior performance and better memory management. In C++ [International Organization for Standards 2017] the `constexpr` specifier declares that it is possible to evaluate the value of a function or variable at compile time [Dos Reis and Stroustrup 2010]. The use of `constexpr` is limited to functions that operate on literal types, i.e., types which are *sufficiently simple* so that the compiler knows their layout at compile time. This includes scalar types, reference types, arrays of literal types and class types with all data members of literal types and `constexpr` member functions and constructors. The compiler checks the validity of the programmers' intention and evaluates it. Our system enables CTFE for Java. To the best of our knowledge, none of the above systems allow code that runs at compile time to utilize analysis results of the compiler to, e.g., customize data structures based on those results. Executing the compile-time code on the Java VM that builds the native image also ensures that compile-time code runs fast.

Process forking is an operation where a process creates a copy of itself. This is usually combined with copy-on-write techniques [Rashid et al. 1987] to avoid copying of physical memory. In VMs this technique can be used to optimize start-up by avoiding running the same expensive initialization multiple times. Unlike heap snapshotting, forking is purely a run-time technique. The Android runtime speeds up application launching by forking the *zygote* process [Ehringer 2010]. Zygote starts execution during the Android startup and preloads Java classes and shared resources. All other application processes start as a fork from the original process inheriting its memory and resources using copy-on-write sharing. However, the process is application independent, i.e., it cannot be used to share application-specific objects and classes; and it cannot be used to pre-initialize the application and therefore cannot improve the load time of the application.

Kawachiya et al. [Kawachiya et al. 2007] clone the state of a running Java VM in a similar way to snapshots. The cloned VM duplicates the heap and dynamically compiled machine code. While the idea can be implemented using the *fork* system call, it requires extensions to the Java VM since *fork* does not duplicate several operating system resources such as threads, mutexes, and file management structures. The Multi-Tasking Virtual Machine [Czajkowski 2000; Czajkowski et al. 2002; Heiss 2005] proposes methods for data sharing among Java VMs. Application isolation techniques allow multiple applications to share the same VM process and reduce individual application footprint and start-up time. Class Data Sharing [Sun Microsystems 2004] and Shared Classes [Corrie 2006] are used in production Java VMs to share class data structures between multiple Java processes.

Lion et al. [Lion et al. 2016] propose reducing the warm-up overhead by reusing JVMs between similar jobs. Their analysis yields that most of the warm-up time is spent loading classes and interpreting bytecodes. By reusing JVMs the cold start-up is avoided: the application can immediately start executing previously loaded and JIT-ed code. The warm-up time is consistent under various workloads, thus it affects short running jobs more. The overhead becomes more significant in systems that abide by the principles of parallelization, i.e., speeding up long-running jobs by parallelizing them into short tasks. Although this approach can improve start-up it reduces application security. Sharing warmed-up VMs between multiple unrelated apps can enable the partial reconstruction of the execution path of prior runs via timing side-channel attacks. Moreover, data from previous runs can leak unless the heap space is zeroed between runs. Our system yields many of the same benefits. AOT compilation avoids bytecode interpretation and class loading at run time. The homogeneity of parallel data-processing jobs enables efficient PGO, therefore ensuring peak performance. Data-parallel jobs can be speed-up by starting a fresh native image or an isolate for each task. Additionally there is no risk of leaking sensitive information between runs since each native image execution starts from the task-independent, statically computed heap.

The Ovm [Armbruster et al. 2007] Real-time Java VM shares some design restrictions with native image. It is implemented in Java, with extended semantics to express low-level operations. It is AOT compiled and an executable image is generated for a particular Java application. The compiler performs a Reaching Types Analysis (RTA) to discover the call graph of the application and to prune dead methods and classes. The RTA operates under a closed-world assumption and it requires developers to provide a list of all reflective methods that may be invoked.

8 CONCLUSIONS

In this paper, we presented a novel approach for compiling a Java application that adheres to a closed-world assumption into a native image. The application can still use Java's modularity concepts such independent library .jar files and service loaders. Our approach produces a lightweight, self-contained executable by iteratively running points-to analysis, initialization code, and heap snapshotting, followed by AOT compilation. Using several case studies and examples, we showed how application initialization at build time and the image heap lead to fast startup, low memory usage, and good peak performance. We believe that our approach makes languages such as Java a viable choice for microservices and serverless cloud functions.

ACKNOWLEDGMENTS

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. We especially thank Mario Wolczko for feedback on this paper.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

REFERENCES

Ole Agesen and David Ungar. 1994. Sifting out the Gold: Delivering Compact Applications from an Exploratory Object-oriented Programming Environment. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 355–370. https://doi.org/10.1145/191080.191135

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 923–935.

Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. 1999. Implementing Jalapeño in Java. In *Proceedings of the ACM*

SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. ACM Press, 314–324. https://doi.org/10.1145/320384.320418

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417. https://doi.org/10.1147/sj.442.0399

Amazon Web Services, Inc. 2019. AWS Lambda. https://aws.amazon.com/lambda/

Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. 2007. A Real-time Java Virtual Machine with Applications in Avionics. *ACM Transactions on Embedded Computing Systems* 7, 1 (2007), 5:1–5:49. https://doi.org/10.1145/1324969.1324974

Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash A. Uzmi. 2010. Optimized Java Binary and Virtual Machine for Tiny Motes. In *Distributed Computing in Sensor Systems*. Springer-Verlag, 15–30. https://doi.org/10.1007/978-3-642-13651-1_2

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 52:1–52:27. https://doi.org/10.1145/3133876

Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In *Proceedings of the European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 17:1–17:31. https://doi.org/10.4230/LIPIcs.ECOOP.2018.17

D.G. Bobrow, D.L. Murphy, and W. Teitelman (Eds.). 1969. *The BBN Lisp System Reference Manual*.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 241–250. https://doi.org/10.1145/1985793.1985827

Rodrigo Bruno and Paulo Ferreira. 2016. ALMA: GC-assisted JVM Live Migration for Java Server Applications. In *Proceedings of the 17th International Middleware Conference*. ACM Press, 5:1–5:14. https://doi.org/10.1145/2988336.2988341

John M. Chambers. 2008. Software for Data Analysis: Programming with R. Springer Publishing Company, Incorporated, Chapter 2.2: An Interactive Session, 13–19.

Ben Corrie. 2006. Java Technology, IBM Style: Class Sharing. https://www.ibm.com/developerworks/java/library/j-ibmjava4/

CRIU. 2019. Checkpoint/Restore in Userspace. https://criu.org

Grzegorz Czajkowski. 2000. Application Isolation in the Java Virtual Machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 354–366. https://doi.org/10.1145/353171.353195

Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. 2002. Code Sharing Among Virtual Machines. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 155–177. https://doi.org/10.1007/3-540-47993-7_7

D Language Foundation. 2019. D programming language. https://dlang.org

Jeffrey Dean, Craig Chambers, and David Grove. 1994. Identifying Profitable Specialization in Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM Press, 85–96.

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM Press, 37–48. https://doi.org/10.1145/1029873.1029879

Gabriel Dos Reis and Bjarne Stroustrup. 2010. General Constant Expressions for System Programming Languages. In *Proceedings of the ACM Symposium on Applied Computing*. ACM Press, 2131–2136. https://doi.org/10.1145/1774088.1774537

David Ehringer. 2010. The Dalvik Virtual Machine Architecture. http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf

A. D. Falkoff and K. E. Iverson. 1973. The Design of APL. *IBM Journal of Research and Development* 17, 4 (1973), 324–334. https://doi.org/10.1147/rd.174.0324

FasterXML. 2019. Jackson. https://github.com/FasterXML/jackson

Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. 2009. Demystifying magic: high-level low-level programming. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 81–90. https://doi.org/10.1145/1508293.1508305

Google. 2012. V8 JavaScript Engine. http://code.google.com/p/v8/

Google. 2019a. Dart Language. https://www.dartlang.org/

Google. 2019b. Dart Snapshot. https://www.dartlang.org/articles/snapshots/

Google Cloud Platform. 2019. Cloud Functions - Serverless Environment to Build and Connect Cloud Services. https://cloud.google.com/functions/

Yang Guo. 2015. Custom startup snapshots. https://v8.dev/blog/custom-startup-snapshots

Janice J. Heiss. 2005. The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM. https://www.oracle.com/technetwork/articles/java/mvm-141094.html

Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 54–61. https://doi.org/10.1145/379605.379665

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 32–43. https://doi.org/10.1145/143095.143114

IBM. 2019. Cloud Functions. https://www.ibm.com/cloud/functions

Daniel H. H. Ingalls. 1983. The Evolution of the Smalltalk Virtual Machine. In *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner (Ed.). Addison-Wesley Longman Publishing Co., Inc., Chapter 2, 9–28.

International Organization for Standards. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization.

Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2007. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 1–11. https://doi.org/10.1145/1254810.1254812

Stefan Koch. 2017. The New CTFE Engine. https://dlang.org/blog/2017/04/10/the-new-ctfe-engine/

Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 507–518. https://doi.org/10.1109/ICSE.2017.53

David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 383–400.

Morten Marquard and Bjarne Steensgaard. 1992. *Partial Evaluation of an Object-Oriented Imperative Language*. Master's thesis. DIKU, University of Copenhagen.

Kevin Marquet, Alexandre Courbot, and Gilles Grimaud. 2005. Ahead of Time Deployment in ROM of a Java-OS. In *Proceedings of the International Conference on Embedded Software and Systems*. Springer-Verlag, 63–70. https://doi.org/10.1007/11599555_9

Microsoft Azure. 2019. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/

Object Computing. 2019. Micronaut. http://micronaut.io

Oracle. 2019a. GraalVM. https://www.graalvm.org/

Oracle. 2019b. Helidon. http://helidon.io

Perf Wiki. 2019. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page

Artur Pilipenko and Oleg Pliss. 2018. Redundancy Elimination in the Presence of Split Class Initialization. In *Proceedings of the International Conference on Managed Languages & Runtimes*. ACM Press, 3:1–3:14. https://doi.org/10.1145/3237009.3237014

Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-driven Incremental Inline Substitution Algorithm for Just-in-time Compilers. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 164–179.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*. IEEE Computer Society Press, 31–39. https://doi.org/10.1145/36206.36181

Derek Rayside and Kostas Kontogiannis. 2002. Extracting Java Library Subsets for Deployment on Embedded Systems. *Science of Computer Programming* 45, 2-3 (2002), 245–270. https://doi.org/10.1016/S0167-6423(02)00059-X

RedHat. 2019. Quarkus. http://quarkus.io

Ian Rogers and Dave Grove. 2009. The Strength of Metacircular Virtual Machines: Jikes RVM. In *Beautiful Architecture*, Diomidis Spinellis and Georgios Gousios (Eds.). O'Reilly, Chapter 10.

Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 126–137. https://doi.org/10.1007/3-540-36579-6_10

Ulrik Pagh Schultz. 2001. Partial Evaluation for Class-Based Object-Oriented Languages. In *Proceedings of the Symposium on Programs As Data Objects*. Springer-Verlag, 173–197. https://doi.org/10.1007/3-540-44978-7_11

Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic Program Specialization for Java. In *ACM Transactions on Programming Languages and Systems*. ACM Press, 452–499. https://doi.org/10.1145/778559.778561

Werner Schuster. 2011. The Essence of Google Dart: Building Applications, Snapshots, Isolates. https://www.infoq.com/articles/google-dart/

Amin Shali and William R. Cook. 2011. Hybrid Partial Evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 375–390. https://doi.org/10.1145/2048066.2048098

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

Shaun Smith. 2018. Announcing Oracle Functions. https://blogs.oracle.com/cloud-infrastructure/announcing-oracle-functions

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Springer-Verlag, Chapter Alias Analysis for Object-oriented Programs, 196–232. https://doi.org/10.1007/978-3-642-36946-9

Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM Press, 165–174. https://doi.org/10.1145/2544137.2544157

Guy L. Steele, Jr. and Richard P. Gabriel. 1993. The Evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*. ACM Press, 231–270. https://doi.org/10.1145/154766.155373

Sun Microsystems. 2004. Class Data Sharing. https://docs.oracle.com/javase/1.5.0/docs/guide/vm/class-data-sharing.html

Warren Teitelman (Ed.). 1974. *Interlisp Reference Manual.*

H. S. Teoh. 2017. Compile-time vs. compile-time. https://wiki.dlang.org/User:Quickfur/Compile-time_vs._compile-time

The Apache Software Foundation. 2019. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html

Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. 1995. Practical Experience with an Application Extractor for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 292–305. https://doi.org/10.1145/320384.320414

Ben L. Titzer. 2006. Virgil: Objects on the Head of a Pin. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 191–208. https://doi.org/10.1145/1167473.1167489

Ben L. Titzer, Joshua Auerbach, David F. Bacon, and Jens Palsberg. 2007. The ExoVM System for Automatic VM and Application Reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 352–362. https://doi.org/10.1145/1250734.1250775

David Ungar. 1995. Annotating Objects for Transport to Other Worlds. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 73–87. https://doi.org/10.1145/217838.217845

James G. Wheeler. 1981. Improved Sharing of APL Workspaces and Libraries. In *Proceedings of the International Conference on APL*. ACM Press, 327–334. https://doi.org/10.1145/800142.805382

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 30 (2013), 24 pages. https://doi.org/10.1145/2400682.2400689

Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. 2017. One Compiler: Deoptimization to Optimized Code. In *Proceedings of the International Conference on Compiler Construction*. ACM Press, 55–64. https://doi.org/10.1145/3033019.3033025

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 662–676. https://doi.org/10.1145/3062341.3062381