Leyden: Lessons from Graal Native

Summary

- We discuss some of the architectural decisions that inform the design of GraalVM Native.
- We explain how they combine to achieve low footprint and fast startup for static java apps.
- We recommend a similar approach for Project Leyden.
- We sketch specific extensions or modifications to the OpenJDK code base that might achieve a similar outcome.
- We conclude by suggesting several possible experiments to assess feasibility of these proposed changes.

Introduction

OpenJDK subproject Leyden was created to enable OpenJDK to deliver ‘static’ Java applications i.e. self-contained Java binaries that:

- include all the code needed to run an application, compiled to native machine code
- do not require an interpreter, compiler or any class loading capability
- can be deployed without any accompanying application or JDK runtime class bytecode

The GraalVM Native Image Generator (GNIG) already addresses this requirement, although, absent any definitive specification of how ‘static’ Java applications are expected to behave, GNIG is of necessity an *ad hoc* solution.

GNIG offers some very appealing benefits as a deployment vehicle when compared with the ‘dynamic’ OpenJDK JVM, in particular reduced footprint for code and metadata and faster startup. These benefits also would be highly desirable for Leyden. However, there are some reasons why GNIG is an unsatisfactory solution for static Java apps. These include political and legal concerns as much as technical ones.

In the latter camp, the most significant issue is the complexity of the design and implementation. That complexity is in part technical debt, accumulated on the meandering path GraalVM followed from a research project towards a product release. However, a lot of its complexity comes from a deliberate choice to address several competing product needs in a single code base. GNIG is only one component of a much larger project.

This does not imply that Leyden cannot profit from considering some of the lessons learned in implementing GNIG. There is good reason to review and perhaps adopt some of the architectural choices it makes. This document considers some of the more useful strategies used by GNIG to transform a Java app into a native image and identifies some valuable opportunities and benefits that follow from adopting them. It concludes by recommending several experiments to assess the feasibility of adopting these strategies.

We do not here attempt to present a comprehensive review of GraalVM, good or bad; this document is tightly focused on what might be useful for Leyden. We present what is needed to consider how similar choices might make sense in the context of Leyden and how far the same opportunities and benefits might be available in a static Java image generator derived from OpenJDK.

Scope

This discussion considers the way GraalVM uses JVMCI and the configuration capabilities of the GraalVM JIT to manage:

- a PointsTo analysis identifying an app's class/method/field reference closure plus static init dependencies between referenced classes
- the minimization of that reference closure using a high-level compiler pass to fold build-time static field values and/or other build-time constants into method code and eliminate references along dead code paths
- the transformation of the JVMCI model of that closure (by wrapping with alternative JVMCI types) to elide redundant classes/methods/fields
- feeding this transformed class model to the Graal Native Compiler to generate method code (GNC — i.e. the Graal compiler configured to perform the same high level pass and with a back end that emits code for a stand-alone image).
- writing the metadata and some heap data into a generated image section where generated method code and Substrate VM runtime code can link to it directly (i.e. using the capabilities of the image linker to achieve direct address linkage, as opposed to having the runtime load this data from an external data source and either explicitly relocate address references at load time or rely on indirect addressing.)

Having discussed each of these in the context of GraalVM they are each reconsidered from the point of view of Leyden. This involves identifying to what degree an equivalent capability either already exists in OpenJDK or could be added by modifying existing OpenJDK components. Where possible, an experiment is suggested to investigate further the feasibility of making any envisaged modifications and to provide a basis for estimating the amount of work that would be involved.

There are many strong reasons for Leyden to utilize existing Hotspot components, or some variant of them modified to cater for the needs of static Java apps. These include keeping the code base minimal, sustaining the benefits of existing knowledge and experience, maintainability, supported architectures, and security.

The GraalVM project's decision to implement a lightweight equivalent to Hotspot in Java (SubstrateVM) is touched on tangentially but not considered as an option for Leyden. Similarly, the use of an alternative JVMCI-based compiler is regarded as out of scope. If nothing else, the significant resources and elapsed time that would be needed to implement a functional, reliable and performant Java VM or compiler rules those options out.

GraalVM Native Image Generator

JVMCI

JVMCI was developed primarily as an interface that allows a ‘foreign’ JIT compiler to be plugged into OpenJDK, whether as the sole compiler or as a second tier compiler. It provides an API which, in its most reduced form, allows a JIT to be registered, to receive compilation requests and to return compiled method code buffers. The API also supports more detailed interactions between the JIT and JVM, e.g. notification of compiled method de-optimization, such as a speculative guard failure at some associated code location.

Since the GNIG does not actually require or provide the Graal Compiler as a JIT for use by the host JVM on which it runs, one might legitimately wonder why GNIG needs to use JVMCI? That is because JVMCI also includes an API to provide a JIT with details of the loaded class base, beyond what is available from the reflection API, and details of the underlying JVM architecture. A JIT needs these details: they provide a complete, self-contained model of a method’s call tree, its immediate and inlined method bytecode, and the hierarchy of types needed by each compilation request. The JVMCI model is also searchable, allowing the whole, resolved class base to be accessed.

The JVMCI metadata model is used uniformly across the whole JVMCI API. It includes information missing from the standard, reflective model of the loaded class base, such as details of class pool entries, method bytecode, exception tables, local variable tables, and so on. Presenting this information via JVMCI avoids the need for the ‘foreign’ JIT to load, parse and record details from raw bytecode files that have already been loaded and processed by the JVM/JDK runtime. It also ensures that the view of the code base presented to the JIT exactly corresponds to that of the JVM for which it is JIT-compiling code.

The JVMCI metadata model is layered, starting with generic interfaces like JavaType, JavaMethod, JavaField, JavaValue, Constant etc. These generic abstractions are gradually refined through extension and specialization, first with abstract implementations that are specific to the Hotspot JVM and then with subclass implementations that implement behaviours specific to the GraalVM JIT and its various GraalVM clients, including GNIG.

For example, we have the following hierarchy for types:

```
interface ResolvedJavaType {
  . . .
}

abstract class HotspotResolvedJavaType implements ResolvedJavaType
{
  . . .
  }

class HotspotResolvedJavaTypeImpl extends HotspotResolvedJavaType
{
    . . .
}
```

A similar hierarchy of generic interface, abstract host-specific class and concrete host implementation class exists for all other metadata elements. So, for example, `ResolvedJavaMethod` instances are passed in compilation requests, giving access to the hosting `ResolvedJavaType` and its related `ResolvedJavaField` and `ResolvedJavaMember` members. Other notable types include `JavaConstant`, `PrimitiveConstant`, `JavaKind` etc.

Further JVMCI metadata Type instances can be obtained using the `JVMCIRuntime` instance obtained during JVMCI startup, either by passing a reflective `Class<?>` instance and retrieving the associated `ResolvedJavaType`, or by resolving a type name against an existing `ResolvedJavaType` to obtain a class-loader-specific instantiation. Once again, this runtime access API is layered. So, the `HotspotJVMCIRuntime` implementation obtained during JVMCI initialization on Hotspot provides these resolved types as instances of `HotspotResolvedJavaType` and they expose their members via types `HotspotResolvedJavaMethod` and `HotspotResolvedJavaField`.

Redefinition via Indirect Metadata

The Graal Compiler consumes instances of the JVMCI metadata model directly, relying on generic interfaces like `ResolvedJavaType`, `ResolvedJavaMethod`, `ResolvedJavaField` and `JavaConstant` for metadata input to the compiler. It does not care whether the underlying implementation is a `HotspotResolvedJavaXXX` or some other implementation. This provides an opportunity for GNIG (and other GraalVM components) to massage the code base prior to compilation by substituting the original suite of `ResolvedXXX` implementation instances with another, indirect implementation that intercepts certain APIs to introduce changes. This is much simpler than the horribly complex alternative of transforming the original bytecode and either: substituting it at the original point of load; or reloading the bytecode, creating an equivalent JVMCI Type and somehow editing it into the model originally presented via JVMCI.

Closing the World with a Points-to Analysis

GNIG's points-to analysis identifies a minimized (but not necessarily *minimal*) set of metadata components needed to compile a native image. It employs a `JVMCIRuntime` instance to iterate over the class base, working from an initial class and main method (or, in the case of a library, a suite of library API methods) and recursively building up a Universe of referenced metadata. Along the way it also verifies that the application satisfies a closed-world hypothesis, constraining the targets for indirect metadata references that arise from use of reflection, method handles and class loading, rejecting cases where no such constraint can be identified.

As the analysis proceeds, each referenced `HotspotResolvedJavaType` is wrapped with an `AnalysisType`, the latter also an instance of `ResolvedJavaType`. `AnalysisType` instances are assembled in an `AnalysisUniverse` which represents a complete transitive closure of all metadata 'reachable' from the starting point. However, the plasticity of the `ResolvedXXX` model allows the meaning of 'reachable' and the resulting model to be subtly and gradually varied relative to the input `HotspotResolvedXXX` model as the analysis proceeds. This provides an extra fine-tuning option to 'edit' the class base, beyond the redefinition by dropping metadata elements that the wrapping model enables.

Class, method, or element substitution

Classes may be explicitly replaced wholesale with a substitute class as they are entered into the `AnalysisUniverse` or they may be replaced element by element with substitute methods or fields derived from another class. The choice of which metadata to substitute and what target metadata to substitute it with is mostly determined by `@Target` annotations supplied on the underlying replacement (target) classes. Target classes reside in GraalVM's own package space and are located in GraalVM-supplied jars. These classes, along with classes supplied in application jars, are pre-loaded and pre-scanned by the GNIC's `ImageClassLoader` prior to initiating the points-to analysis. In some special cases transformations or substitutions are performed using compiler plugins (using something akin to an architecture neutral equivalent of C2 intrinsics).

Analysis metadata instances normally just punt calls to `ResolvedJavaXXX` API methods to the type they wrap. Exceptions to that rule generally perform semantically neutral operations such as normalizing names to satisfy linker naming constraints. However, in certain special cases, the analysis inserts a special wrapped metadata instance between itself and the type it was created to model, an instance of `SubstitutedType`, `SubstitutedMethod` or `SubstitutedField` or one of their subclasses. These are still metadata types, instances of the corresponding `ResolvedJavaXXX` interface, but they serve to edit the code base. They retain links to the originally referenced `HostpotResolvedJavaXXX` instance plus an alternative (target) `ResolvedJavaXXX` instance supplied by GraalVM. The target is usually another `HostpotResolvedJavaXXX` instance derived from a loaded class with an `@Target` annotation. In rare cases the target is a custom `ResolvedJavaXXX` instance defining a synthetic method or field with no associated Java source.

`SubstitutionXXX` instances intercept specific metadata API methods, in particular fielding queries to report their structure (e.g. for a class, to list all methods, return all constant pool elements etc) and selectively returning information derived from either an element of the original or the substituted, aka *target,* metadata instances or some combination of such elements (e.g. swapping some of methods in the original's list with methods read from the target class). This mechanism makes it possible to add, drop or rename types and members or even provide a partially or completely different definition for a metadata element.

Analysis proceeds recursively, which means that substitutions made at one point can direct the loading and replacement of further loaded classes. This allows some elements of the `AnalysisUniverse` to be marked redundant as regards the final image. That may happen because a class or some of its fields or methods are not actually referenced on a live path exercised by the application. Alternatively it may be possible to omit a class because it is only used for static initialization of some other class and the static initialization can safely be performed at build time rather than at runtime.

Partial evaluation and code optimization

The progress of points-to analysis also relies upon the Graal compiler, configured as a partial evaluator, to implement metadata reference reductions at each step where a method is processed. The compiler recursively constructs a method graph from the method's bytecode. It compiles the method graph, performing high level optimizations such as constant inlining, 'intrinsic' method substitution, dead code elimination and type flow analysis but omitting to lower the transformed graph to machine code. This reduction process works very effectively in combination with build-time static initialization, where the compiler treats certain static fields as final, using the values they were initialized to in the hosting JVM, rather than treating them as requiring initialization in the target native image. An optimized graph output by this high-level compile/partial evaluation step can be scanned to derive a much more restricted set of reachable types, fields and methods than would be obtained by simply parsing the input bytecode, enabling more aggressive pruning of the Analysis Universe.

Note that detecting whether build-time init is permissible is both complex and heuristic. It involves a recursive, global scan of all class static init code to detect whether it includes operations which might side-effect or be side-effected by other clinit methods, including those that are known not to be safe to drop at build-time. When build-time class init is discovered to be legitimate then static field values can be treated as constant and generated code can be modified to avoid calling class init methods.

Note also that the Analysis stage has to address the converse issue of initiating runtime static initialization from compiled code. The dynamic JVM performs static initialization as a side-effect of class loading. With GNIG the generated native method code must ensure that static init routines are invoked at an equivalent point during application execution. There may be multiple locations where an init call for some specific class may be needed so call sites are guarded with a check that initialization is still outstanding. The Analysis phase tracks class static init dependencies and uses the resulting dependency graph to avoid planting static init calls in methods where it can be proven that they will never be needed..

The Hosted Universe

The Analysis metadata is wrapped a second time before building and writing the native image and, in particular, feeding methods to the compiler to build the image's code segment. These secondary wrappers include the family of types below `HostedType` (`HostedInstanceType`, `HostedArrayType`, etc), `HostedMethod` and `HostedField`. `HostedXXX` wrappers hide the API needed to perform analysis, instead providing an API appropriate for managing compilation of methods and serialization of code/data to an image file. Instances of the `HostedXXX` types are located in a `HostedUniverse` which wraps the underlying `AnalysisUniverse`. The result is that HostedXXX instances can be uniformly mapped to `AnalysisXXX` instances and, through them, to the underlying `HotspotResolvedJavaXXX` instances or their substituted alternatives. The `HostedXXX` classes are the primary metadata used internally by the Image Builder and Writer classes and by SubstrateVM classes. Since they implement the `ResolvedJavaXXX` interfaces they can also be used as the metadata input to the GNC.

Bottoming out at SubstrateVM

The `AnalysisUniverse` built by GNIG includes Java classes, provided as part of SubstrateVM, which implement JVM functionality. Use of this Java-in-Java VM model requires the translation or substitution of some standard OpenJDK Runtime classes in order to allow for differences between the operation of SubstrateVM and that of a normal dynamic JVM. This goes beyond the substitutions and translations needed to remove or disable dynamic behaviours from standard JDK code (also application code) that are redundant in a static Java app.

SubstrateVM imposes extra requirements on the Graal compiler, which needs to understand a few things about the special semantics of some SubstrateVM code. Some are straightforward things to to do with back-ending to a different VM model, e.g. different layout/location for instance or static fields; or merging VM runtime metadata into the `java.lang.Class` mirror (actually, the proxy class `DynamicHub` that substitutes `java.lang.Class`). These needs are met using the rich configuration options supported by the Graal compiler. For example, the compiler accepts a target VM configuration that allows details of SubstrateVM to be folded into the compilation process, alongside the more conventional configuration options that relate to the target architecture; the compiler front-end is driven by a configurable suite of stages, phases and transformations that can be respecified to include SubstrateVM-specific graph construction and transformation steps; the backend is driven by a lowering provider and lowering classes that can be overridden to introduce code assembly steps specific to SubstrateVM.

Other requirements arise from the use of Java itself to implement VM operations. Some code must be compiled specially or, at least, verified by the compiler to ensure it does not indirectly depend on JVM behaviour it seeks to implement. For example, safepoint processing happens in a Java thread and may allocate objects, so it cannot assume a GC will not happen; code run by a GC must be uninterruptible, so it may not execute synchronizations or object allocations and must be compiled without safepoint checks. Dealing with these latter requirements is done by a mixture of analysis-time and compile-time checks that are not simply variations on the normal operation of the compiler. It adds complexity to the compiler design and implementation that is not needed simply to compile normal application or JDK runtime methods to native code (i.e. that is not needed when relying on a foreign language JVM like Hotspot).

Image Building & Writing

Building a native image requires generating a binary in an OS-native format. This requires writing constant Java data values and initial Java static field values into a data segment, and compiled Java code into a code (aka text) segment at generation time. The data section splits conceptually into read-only and read-write sub-sections, and further into primitive value and object value subsections.

The Analysis stage must track which data is live and which methods need to be compiled. A traversal of the live data can be used to associate each constant primitive value or object and each static field location with an offset and occupied extent in the data section. The compiler can be fed methods one at a time to generate buffers of compiled native code which can also be associated with an offset and occupied extent in the code section.

The GraalVM Image Builder and compiler try to ensure that all cross-references between code/data elements in the generated sections and in foreign C objects can be resolved by the image linker, either by exporting and importing symbols or through the use of segment internal and cross-segment relocations. Most symbols can be resolved when the image is written. Only linkage to symbols in a few operating system dynamic (shared) libraries need to be resolved at link-load time. Relying on the linker and link-loader rather than custom initialization code to establish such linkage offers a significant improvement to startup time as well reducing code footprint.

So, the generator must ensure that references from Java code and data to:

1. other Java code or data or
2. foreign (i.e. C) library code or data

employ an appropriate linkage model. In most cases, references are within the generated sections. Intra-section references may sometimes be encoded using direct offsets without the need for a relocation e.g. PC-relative branches for calls within the generated code section. In other cases, a local segment offset relocation is needed e.g. a call within the code section that needed to use an absolute branch would use such a relocation to rebase an offset to the absolute address, relying on the operating system linker to update the branch instruction when it writes the executable image with the section start assigned a specific VMEM address. References across generated sections can also use segment plus offset-based relocations e.g. generated code section references to a Klass would employ the offset of the Klass in the metadata section and be rebased against that section. Once again these can be relocated to a determinate address when the executable image is written.

References to data and code generated by a 3rd party tool (e.g. to compiled C code) require identification of the target by mention of a symbol, possibly with an associated offset. The symbol must be exported by the target object, allowing its location in the library's text or data section to be identified and converted to a segment based offset and, from there, a target VMEM address during linking of the final image.

In special cases it is necessary for linkage to be introduced by an indirection table that is dynamically populated by custom initialization code during application startup. This is the case for linkage to the operating system dynamic libraries that are loaded using the standard JNI linkage model.

Towards a Leyden Native Image Generator

Using C2 CI as an equivalent to JVMCI

The Hotspot JVM has its own internal model of Java metadata that it builds incrementally as it loads class bytecode. This model is built using instances of C++ classes such as Klass, ArrayKlass, InstanceKlass, Method, ConstantPool, etc. The only significant element of the original class files retained is the method bytecode.

These internal C++ classes include all the structural aspects of the class base that are needed in order to drive a method's high-level compilation, much like the data made available via JVMCI. However, they do not constitute the presentation layer consumed by the compiler because they are VM internal types. The C1 and C2 compilers' front ends access the information in this model via the CI interface, which isolates them from details of the VM's underlying implementation. For the most part the CI interface successfully provides an isolation barrier.

The CI API could be used much like JVMCI to allow a transformed and/or substituted model of a loaded Java class base to be presented to C2 for compilation to native code. Wrapper implementations of the CI Class, Method and Field interfaces could selectively replace elements of the code base derived from loaded bytecode with alternative elements that add, remove or substitute behaviour. Those alternative elements could themselves be derived from other loaded class bytecode, using a target model similar to Graal, or they could be synthesized using a suitable declaration language or custom code.

Driving Static Compilation from Java

One option for compiling Java methods would be to drive the C2 compilation process from Java much as is done by GNIG, using JVMCI as the data model that underpins the CI wrappers. A Java program could read the JVMCI model of the loaded class base, transform it as part of a points-to analysis and then present JVMCI Method instances back to the JVM for static compilation via C2. The resulting code could then be written into an image code section alongside the required metadata and instance data.

Indeed, if C2 were exposed via an (internal-facing) JVMCI backend then the standard backend compile API could be used as the channel for compilation requests, generating the code into a buffer provided from Java and returning all the required auxiliary info defined in the backend API.

In order for this to work, the JVM would need to implement a protocol bridge to convert JVMCI data fed through the compile API into CI data that C2 can consume. It would need to provide a suite of CI wrapper types which would redirect CI API calls to the underlying JVMCI Java instances and in turn wrap any returned JVMCI data.

One element is missing from this option. It does not provide a way to perform the high-level compilation steps that GNIG uses to optimize the reference closure analysis. It might be possible to add a new backend API for this purpose, taking a JVMCI Method in and returning a suite of JVMCI metadata objects that are referenced from the input method. This would also require C2 to be extended to support a high-level compilation API that would take an input method and associated metadata, compile it to an optimized high-level graph, and retrieve all remaining references to CI metadata from the graph as roots for recursive derivation of the reference closure.

Driving Static Compilation from C++

An alternative approach would be to conduct the points-to analysis within libjvm.so itself. Initial CI instances for the entry class and method would be constructed directly over the internal metadata model. Wrapper instances could be created by the points-to analysis as it identified elements that needed substitution, whether by replacement, translation or deletion.

This would avoid much of the extra complexity of translating backwards and forwards between C++ and Java representations, and also avoid the memory costs involved in performing those translations. It would also probably make it easier to open up an internal ‘whitebox’ API to the compiler to perform method graph optimizations during the points-to analysis.

Once the CI Analysis Universe has been constructed it might still be useful to expose the contents of this Universe to a Java program via JVMCI wrappers (or using the standard JVMCI instantiation in cases where the CI model was left untransformed). This would allow the rest of the Leyden Native Image Generator (LNIG) to be implemented in Java. Methods included in the closure could be passed back to C2 for compilation. The resulting code could be written into a code section alongside the required metadata and data sections.

Runtime Static Init

Static init methods for classes embedded in a native image need to be run at a point where an equivalent execution would happen when executing the code on the dynamic JVM. The C2 compiler will need to insert guarded static init calls into compiled code that does not have some clear exemption to indicate that init will already have been performed. The Leyden points-to analysis will probably need to perform an analysis of static init dependencies in order to avoid the cost of redundant static init checks.

A few runtime classes must be pre-initialized before any Java thread can start executing, just as some classes are pre-initialized during dynamic JVM bootstrap. It would be beneficial to performance if this pre-initialization could be done at build time on Leyden rather than during image startup and if a larger subset of classes than the ones managed in dynamic JVM initialization could be included in the set that are pre-initialized.

Build-Time Static Init

[Need to investigate what is possible *and* what is legitimate here. This is a problematic topic and one that Leyden needs to address as a matter of policy not feasibility.]
The introduction of build-time static initialization can help a Points-To Analysis to produce a much smaller reference closure and open up extra opportunities for optimizations when compiling methods in that closure. This may serve to significantly reduce footprint and startup time but it also comes with many potential perils. Hard-wiring values that would

normally be initialized during program execution may significantly belie the specified semantics of the Java program. Even if the substitution of constant data for computed data is legitimate it can also introduce different execution ordering, computing unexpected results, including results that are not necessarily appropriate to the runtime context. So, any Leyden implementation that supports build-time initialization will need to derive its behaviour from a clear specification of what sorts of static data it is legitimate to pre-compute and to provide some way of verifying that any such data have been computed correctly.

C2 Native Configuration

If the Leyden image generator program is implemented in Java but expects C2 to generate target image code then this raises the issue of what configuration to use for host runtime vs native target compilation tasks. The latter may need to be configured appropriately for the target architecture, possibly using different compiler options to those employed in the host JVM. The current configuration options are all provided as single-setting global variables. It may be necessary to refactor these into a default (hosted) compiler context and an (alternative) native compiler context and modify the C2 code so that it makes decisions based on values in the current context.

C2 Code, Metadata and Object Data Linkage

C2 generated code needs to be able to refer to metadata and to objects such as String constants and java.lang.Class mirrors, as well as primitive data values. So, generating static code ahead of time in a code program section also requires generating a metadata and initial heap section containing these elements. The generator needs to ensure that these sections are correctly linked when they are loaded into a running process image. It also needs to ensure that the metadata and object data can be referenced (as needed) from the Hotspot C++ code compiled into the Leyden version of libjvm.so.

The generated method code produced by C2 under either of the above compilation schemes would need to employ a different linkage model to code generated by C2 for execution in a host JVM. Ahead-of-time generated code has to be embedded in a program section that is ‘statically’ linked both internally, from one compiled method to another, and also externally, to a generated metadata section, to objects such as class mirrors generated into an initial heap section, and to foreign C/C++ code and data exported by the static JVM native and Hotspot libraries. Normally such an executable code section would be expected to be read only so the linkage must either be resolved and fixed when building the image or else must go via indirect tables encoded in a separate writable section.

JVM code, by contrast, employs a writable code section, allowing it to patch runtime code and data addresses for these targets directly into code buffers or their associated constant pools and retaining the option of repatching a new target address. Adapting C2 for Leyden will require changing the C2 back end to use different, more appropriate forms of relocatable reference. These should allow linkage to be established wherever possible using the relocation model supported by the image build-time linker and the operating system link-loader.

Relocations will be required to ensure the correct runtime addresses are employed for branches to call targets, loads of metadata constants and loads of constant objects and primitive values. As with GraalVM most references should be able to be resolved to fixed addresses by the time the final executable image is written.
Call target addresses will never change in a static image so branch addresses can be embedded directly in compiled code. Call linkage for Java calls can probably be achieved using a PC relative branch without the need for a relocation. Alternatively, if the code cache size exceeds the address range available for offset based branches they can employ an absolute branch with a local section + offset relocation used to update the target address during executable image linking. Call linkage for runtime method calls (i.e. branches from the generated code section to compiled C++ routines exported by the text section) can use an absolute branch employing an extern symbol-based relocation to update the target address during executable image linking.

Metadata object addresses will also never change in a static image so it will be possible to embed them directly in code, again using a section + offset relocation to update the target address during image linking. This avoids the need to embed metadata addresses in a constant pool associated with the compiled method.

Object constants in the initial heap section are not guaranteed to remain at the same address. If they can be moved at GC then it will be necessary to load object and primitive constants from a constant pool associated with the method. Assuming the generated code section is made read only this implies that constant pools (or at least the parts of them that include object references) will need to be stored in a separate writable section. That further implies that loads of constant pool slot addresses will need to employ section + offset relocations to update the target address during image linking.

The Hotspot VM expects compiled code to be represented as a `CodeHeap` comprised of `CodeBlob` instances which embed the method code and related data in distinct segments which make up the address range covered by the `CodeBlob`. Individual methods are modeled as an instance of C++ class `CompiledMethod` and its subclass `nmethod`. The method subclasses include a variety of data, not just the code buffer containing the actual generated machine code but also information about the load and compile state of the method, constant pool data etc, some of which is updated during the lifetime of the method.

Much of this data will be redundant in Leyden and in large part it should be read-only. It will be necessary to generate a similar form of code indexing structure in order to allow the Leyden runtime to perform tasks like mapping PC addresses to methods, locating live stack slots in stack frames, traversing constant pool objects as GC roots, etc. However, it is very likely that the Leyden Hotspot code will be able to use a simplified code indexing scheme. Also, rather than embed all these elements in one single code heap section it would probably be better to add the generated code to a read-only code section and store the rest of the method info in the metadata section.

It may be worth considering pinning at least some of the immutable initial heap objects and constant data values, for example `Class` and `ClassLoader` mirrors, constant `String` instances and numeric constants. That would allow addresses for these objects to be embedded directly in generated code, avoiding the need to load them indirectly via a constant pool. Pinning Class mirrors has the added benefit of pinning static class fields (these are embedded at the end of the `Class` instance) which should improve the performance of static field reads and writes.

An image writer will need to ensure that all the required Object and primitive data is written to a generated initial heap section. This may include object networks reachable from static fields initialized at build-time. This will require the writer to size all the items to be written and compute a layout for the code and metadata/heap data sections before writing any content. C2 references to data/metadata in these sections can then be converted to linker binary format relocations that specify a section base + offset. C2 references to foreign code or data targets can be specified using symbol-based linker relocations.

ClassLoader mirrors present a specific problem in a static Java native image that does not occur with other metadata mirrors belonging to classes like Class, Module etc. Classloader is the only part of the mirror class suite that is extensible by application defined subclasses (for sure?). Class loaders cannot be functional in a native image so the core functionality of any class loader will be rendered redundant by making the JVM implementation of class lookup calls fail rather than return a Class not already defined in the image. However, an application defined classloader may still want to cache state and implement behaviours relevant to the operation of the app. It may not be possible to pre-cache some Object state of the ClassLoader mirror if, say, it stores data that is specific to the generate-time environment -- such as open file handles -- or is computed from generate-time static data that needs to be re-initalised at runtime -- such as security keys. This will probably require handling using some sort of substitution mechanism similar to that provided by GraalVM to avoid inclusion of redundant state and/or modify behaviour to recompute it.

Not all of the code generated by Hotspot is produced by the C2 compiler. There are also runtime stubs generated into a CodeBlob using direct calls to the Assembler. Generating these stubs at build-time will be important for ensuring quick image startup and will ensure that all generated code resides in a single, read-only code section. So, these stubs should also be written into an image code segment and referenced from JITted Java code using intra-section linker code relocations. Note that many stubs (e.g. interpreter and deopt stubs) will not be needed and it may be possible to simplify those that are required.

In a few cases stubs generated into the code section will need to be visible to Hotspot code (for example the exception return stub must be accessible to code that handles a SEGV caused by a null pointer dereference). In such cases the address of the stub will need to be exported using the linker symbol that the Hotspot code expects to resolve against. A less preferable alternative would be to provide a Leyden JVM with custom init code that installs the target address in a location where Hotspot expects to access it.

A Static Hotspot Core VM + Runtime

With a metadata, code and initial heap generation model as outlined above the Hotspot JVM should not need significant modification in order for it to be capable of being linked statically into a Java native image. Some parts of the existing code base are redundant (none of the interpreter, c1, opto and asm code is needed in a static runtime as is most of the classfile code). Most of the remaining parts of the runtime that need significant changes will require avoiding calls to initialize C++ and Java data structures or generate code heap management data and contents -- because has already been created and embedded into the metadata, initial heap data and code sections to which libjvm.a must be linked.

Some parts of the JVM initialization code may need to be modified to provide slightly different behaviour (for example, the garbage collectors and the underlying memory region mapping code will need to be able to include the initial heap area into its set of managed regions). Other parts of the runtime may need modifications to avoid following code paths that are only legitimate when an interpreter and compilers are present (for example, frame management code will need to be able to avoid case handling that would normally be executed to deal with interpreter frames). However, a great deal of the existing Hotspot JVM code can link to and support execution of pre-compiled Java code using pre-defined Java metadata and initial heap data without change.

In many ways it would be simpler and cleaner if the obviously redundant parts of the JVM code base were compiled out of the static libjvm.a, either wholesale removal of classes or partial removal of specific implementation and API methods and fields. Given the complexity of the code base that might well lead to a very large number of localized changes. In which case it might be better to leave some code and data in the library even though it might not be needed.

Reduction or Removal of Metadata and Related Structures

Several reasons for the compiler to be aware of metadata layout disappear in a closed world static image. Without profiling, deoptimization, and recompilation there is no need for a ConstantPoolCache or MethodData. There is no interpreter, or interpretation-time linking. Dynamic evolution of the class base cannot happen. Speculative compilation is still an option but it would have to include a statically-generated fallback path. Many virtual calls can be turned into direct ones, allowing associated vtables to be pruned. In some cases, execution that relies on vtables and itables or testing of the super and interface hierarchy might profitably be reduced to generated case switch tests.

So, some metadata and related structures that need to be queried to guide execution will be present in the image. However, much of this data can be read-only, and a lot of it can be pruned. For example, there may be no need to lock and update dictionaries attached to class loaders because the resolution of names to class loaders and classes will be predetermined. Furthermore, in many cases class lookups can be converted to explicit, direct class references from the generated code. There may still be a need to be able to search for classes by name or Symbol and classloader, retrieving an associated Klass, but it may be possible to implement this using a reduced type with a much simpler API.

Build-time Metadata Relocation and Serialization

An image writer will need to serialize metadata structures for all classes in the application reference closure into an object file. In order for these metadata objects to be usable from compiled Hotspot VM code the layout of each object needs to conform to the equivalent C++

object layout (n.b. that includes ensuring that objects with virtual methods have their vtable slots correctly linked to their respective vtables). Global roots into the metadata (e.g. the head of the class loader data graph or special classes like `Class`,`String`,`Throwable` etc) will need to be exposed as symbols. In many cases it should be possible to do this using linkage names that conform to those of corresponding C++ static fields employed in the Hotspot code, ensuring that Hotspot code can link automatically without the need to patch up references.

In order for this relocation to work, when building libjvm.a for use in a Leyden image Hotspot code will need to be compiled conditionally to omit definition and initialization of the corresponding static fields. Effectively, the declaration and initialization have been relocated to the generated code.

In the case of data whose address needs to be explicitly inserted into a JVM structure that is managed by the Hotspot code rather than referenced symbolically as a static field value it may be necessary to compile Leyden-specific initialization code into libjvm.a. This latter case will require export of the relevant generated metadata element via some newly introduced symbol. Custom init code will be needed to explicitly link against such symbols and insert the referenced data into JVM structures. Runtime patch-ups like this should be the exception not the norm.

As an example, the list of well known classes exposed via C++ class vmClasses employs a static array `static InstanceKlass* _klasses[];`. In the dynamic VM InstanceKlass pointers are inserted into this array during startup as Java definitions are loaded from bytecode. In the static VM the array could be written into the metadata segment and exposed using the relevant C++ static field symbol. Alternatively, the array could be initialized during JVM startup with Leyden-specific code that employs symbolic references to each well known InstanceKlass in the metadata section. The latter option would require the generator to track these classes during serialization and generate symbols which advertise their location.

Which Dynamic JVM Metadata to Serialize in a Reduced Static Layout?

As well as the immediate metadata objects like Klass, Method etc, it will be necessary to serialize many auxiliary objects referenced from these objects; such as stack maps, instance and class GC maps, line number tables, etc. In order to draw a line around the suite of data included in the generated section it will also be necessary to work back upwards and include some objects which reference the above metadata items and make them accessible to Hotspot code. For example, Klass instances are all indexed from a linked list of ClassLoaderData objects and the head of that list is one of the main roots via which Hotspot code like the GC accesses Klass instances. Similarly, the global symbol table needs to be included so that symbols can be looked up from Hotspot code.

The code running in the host dynamic JVM that writes out the metadata objects will need to distinguish elements needed for the dynamic runtime from the subset that is required for the static runtime. So, it will need to be provided with a description of which target metadata content and layout as needed for use in the static libjvm. This is a tad more tricky to automate in C++ than in Java. GraalVM uses reflection and annotations to identify which fields are needed and remap offsets from one format to the other. Something similar could be done in C++ using macros ranging over the dynamic type and a correspondingly reduced static type but that might require them both to be in scope in the JVM runtime that is doing the writing.

Reuse of Class Data Sharing?

The existing Class Data Sharing (CDS) code provides a possible starting point for implementing this metadata (and auxiliary data) serialization. CDS already knows how to lay out class metadata into both a read-only and a read-write segment that the dynamic JVM can link to when it bootstraps. There are several issues that need addressing to reuse some of this code. CDS would need to be upgraded to allow it to omit dynamic JVM data. CDS mostly relies on explicit linkage of data, where the dynamic JVM maps the CDS segment(s) into memory and then explicitly links the classes and other elements it contains into its runtime class base during bootstrap. This is necessary in order to allow classes to be defined as though they had been dynamically loaded and then updated as new code is loaded. By contrast, a static variant of CDS might be able to make a lot more of the metadata read only and expose elements of the metadata using global symbols that directly resolve symbolic references from the native JVM library.

Experiments To Assess Feasibility

1. Build a JVM that omits the obvious dynamic components (interpreter, compilers etc) and see what needs to be changed for it to build (this task is already under way).
2. Relocate static field definitions and init code for data we want to generate into separate compile units which can be built separately from Hotspot then check it still links
3. Implement a CI wrapper type and hand crank a special case substitution for one specific method with an alternative implementation taken from some target class
4. Prototype a Closed World Analysis that operates on CI (without any reduction or substitution) -- could be in C++ or using Java with special JNI/nativeaccess to CI data.
5. Expose a C2 high level compile API that takes CI method in and returns a set of IR references, doing some minimal set of high level transformations on the graph.
6. Investigate changes to C2 back end to generate code in a buffer plus a set of adapted ‘relocs’ that would allow it to be serialized into an ELF text segment.
7. Assess changes needed to make CDS code expose serialized data via C++ symbols for roots and ‘well-known objects’ (this task is already under way).
8. Investigate strategies for implementing build-time static initialization and for verifying correctness or, at least, identifying specific incorrect cases.
9. Modify Hotspot GCs to support a ‘pinned’ object region and allocate mirrors and String constants for bootstrap, module and system loader classes (i.e. classes that are never unloaded) in the pinned area.