This page intentionally left blank

Chapter 8

Accelerating Time to Steady State with OpenJDK HotSpot VM

Introduction

Start-up and warm-up/ramp-up performance optimization is a key factor in Java application performance, particularly for transient applications and microservices. The objective is to minimize the JVM's start-up and warm-up time, thereby accelerating the execution of the application's main method. Inefficient start-up or warm-up performance can lead to prolonged response times and increased resource consumption, which can negatively impact the user's interaction with the application. Therefore, it's crucial to focus on this aspect of Java application performance.

While the terms *warm-up* and *ramp-up* are often used interchangeably, by understanding their nuances, we can better appreciate the intricacies of JVM start-up and warm-up performance. So, it's essential to distinguish between them:

- **Warm-up:** In the context of Java applications and the JVM, warm-up typically refers to the time it takes for the JVM system to gather enough profiling data to optimize the code effectively. This is particularly crucial for JIT compilation because it relies on runtime behavior to make its optimizations.
- **Ramp-up:** Ramp-up generally denotes the time it takes for the application system to reach its full operational capacity or performance. This could encompass not just code optimization, but other factors such as resource allocation, system stabilization, and more.

In this chapter, we first explore the intricate details of JVM start-up and warm-up performance, beginning with a thorough understanding of the processes and their various phases. We then explore the steady-state of a Java application, focusing on how the JVM manages this state. As we navigate through these topics, we will discuss a variety of techniques and advancements that can help us achieve faster start-up and warm-up times. Whether you're developing a

microservices architecture, deploying your application in a containerized environment, or working with a traditional server-based environment, the principles and techniques of JVM start-up and warm-up optimization remain consistent.

JVM Start-up and Warm-up Optimization Techniques

The JIT compiler has seen continuous improvements, significantly benefiting JVM start-up and ramp-up (which includes warm-up) optimizations. Enhancements in bytecode interpretation have led to improved ramp-up performance. Additionally, Class Data Sharing optimizes start-up by preprocessing class files, facilitating faster loading and sharing of data among JVM processes. This reduces both the time and the memory required for class loading, thereby improving start-up performance.

The transition from the Permanent Generation (PermGen) to Metaspace in JDK 8 has positively impacted ramp-up times. Unlike PermGen, which had a fixed maximum size, Metaspace dynamically adjusts its size based on the application's runtime requirements. This flexibility can lead to more efficient memory usage and faster ramp-up times.

Likewise, the introduction of code cache segmentation in JDK 9 has played a significant role in enhancing ramp-up performance. It enables the JVM to maintain separate code caches for non-method, profiled, and non-profiled code, improving the organization and retrieval of compiled code and further boosting performance.

Going forward, Project Leyden aims to reduce Java's start-up time on HotSpot VM, focusing on static images for efficiency. Broadening the horizon beyond HotSpot, we have a major player in GraalVM, which emphasizes native images for faster start-up and reduced memory footprint.

Decoding Time to Steady State in Java Applications

Ready, Set, Start up!

In the context of Java/JVM applications, start-up refers to the critical process that commences with the invocation of the JVM and is thought to culminate when the application's main method starts executing. This intricate process encompasses several phases—namely, JVM bootstrapping, bytecode loading, and bytecode verification and preparation. The efficiency and speed of these operations significantly influences the application's start-up performance. However, as rightly pointed out by my colleague Ludovic Henry, this definition can be too restrictive.

In a broader sense, the start-up process extends beyond the execution of the `Application.main` method—it encompasses the entire initialization phase until the application starts serving its primary purpose. For instance, for a web service, the start-up process is not just limited to the JVM's initialization but continues until the service starts handling requests.

This extended start-up phase involves the application's own initialization processes, such as parsing command-line arguments or configurations, including setting up sockets, files, and other resources. These tasks can trigger additional class loading and might even lead to JIT

compilation and flow into the ramp-up phase. The efficiency and speed of these operations play a pivotal role in determining the application's overall start-up performance.

Phases of JVM Start-up

Let's explore the specific phases within the JVM's start-up process in more detail.

JVM Bootstrapping

This foundational phase focuses on kick-starting the JVM and setting up the runtime environment. The tasks performed during this phase include parsing command-line arguments, loading the JVM code into memory, initializing internal JVM data structures, setting up memory management, and establishing the initial Java thread. This phase is indispensable for the successful execution of the application.

For instance, consider the command `java OnlineLearningPlatform`. This command initiates the JVM and invokes the `OnlineLearningPlatform` class, marking the beginning of the JVM bootstrapping phase. The `OnlineLearningPlatform`, in this context, represents a web service designed to provide online learning resources.

Bytecode Loading

During this phase, the JVM loads the requisite Java classes needed to start the application. This includes loading the class containing the application's `main` method, as well as any other classes referenced during the execution of the `main` method, including system classes from the standard libraries. The class loader performs class loading by reading the `.class` files from the classpath and converting the bytecode in these files into a format executable by the JVM.

So, for example, for the `OnlineLearningPlatform` class, we might have a `Student` class that is loaded during this phase.

Bytecode Verification and Preparation

Following class loading, the JVM verifies that the loaded bytecode has the proper formatting and adheres to Java's type system. It also prepares the classes for execution by allocating memory for class variables, initializing them to default values, and executing any static initialization blocks. The Java Language Specification (JLS) strictly defines this process to ensure consistency across different JVM implementations.

For instance, the `OnlineLearningPlatform` class might have a `Course` class that has a static initializer block that initializes the `courseDetails` map. This initialization happens during the bytecode verification and preparation phase.

```
public class Course {
    private static Map<String, String> courseDetails;

    static {
   // Static initialization of course details
        courseDetails = new HashMap<>();
        courseDetails.put("Math101", "Basic Mathematics");
```

```
        courseDetails.put("Eng101", "English Literature");
        // Additional courses
    }

    // Additional methods
}
```

The start-up phase is critical because it directly impacts both user experience and system resource utilization. Slow start-ups can lead to subpar user experiences, particularly in interactive applications where users are waiting for the application to start. Additionally, they can result in inefficient use of system resources because the JVM and the underlying system need to do a lot of work during start-up.

Reaching the Application's Steady State

Beyond the start up phase, a Java application's life cycle includes two other significant phases: the ramp-up phase and the steady-state phase. Once the application starts its primary function, it enters the ramp-up phase, which is characterized by a gradual improvement in the application's performance until it reaches peak performance. The steady-state phase occurs when the application executes its main workload at peak performance.

Let's take a closer look at the ramp-up phase, during which the application performs several tasks:

- **Interpreter and JIT compilation:** Initially, the JVM interprets the bytecode of the methods into native code. These methods are derived from the loaded and initialized classes. The JVM then executes this native code. As the application continues to run, the JVM identifies frequently executed code paths (hot paths). These paths are then JIT compiled for optimized performance. The JIT compiler employs various optimization techniques to generate efficient native code, which can significantly improve the performance of the application.
- **Class loading for specific tasks:** As the application begins its primary tasks, it may need to load additional classes specific to these operations.
- **Cache warming:** Both the application (if utilized) and the JVM have caches that store frequently accessed data. During ramp-up, these caches are populated or "warmed up" with data that's frequently accessed, ensuring faster data retrieval in subsequent operations.

Let's try to put this in perspective: Suppose that along with our `Course` class, we have an `Instructor` class that is loaded after our web service initializes and obtains its up-to-date instructors list. For that `Instructor` class, the `teach` method could be one of the many methods that the JIT compiler optimizes.

Now take a look at the class diagram in Figure 8.1, which illustrates the `OnlineLearningPlatform` web service with `Student`, `Course`, and `Instructor` classes and their relationships. The `OnlineLearningPlatform` class is the main class that is invoked during the JVM bootstrapping phase. The `Student` class might be loaded during the bytecode-loading phase. The `Course` class, with its static initializer block, illustrates the

bytecode verification and preparation phase. The `Instructor` class is loaded during the ramp-up phase, and the `teach` and `getCourseDetails` methods may reach different levels of optimizations during ramp-up. Once they reach peak performance, the web service is ready for steady-state performance.

The phase chart in Figure 8.2 shows the JIT compilation times, the class loader times and the garbage collection (GC) times as the application goes through the various phases of its life cycle. Let's study these phases further and build an application timeline.

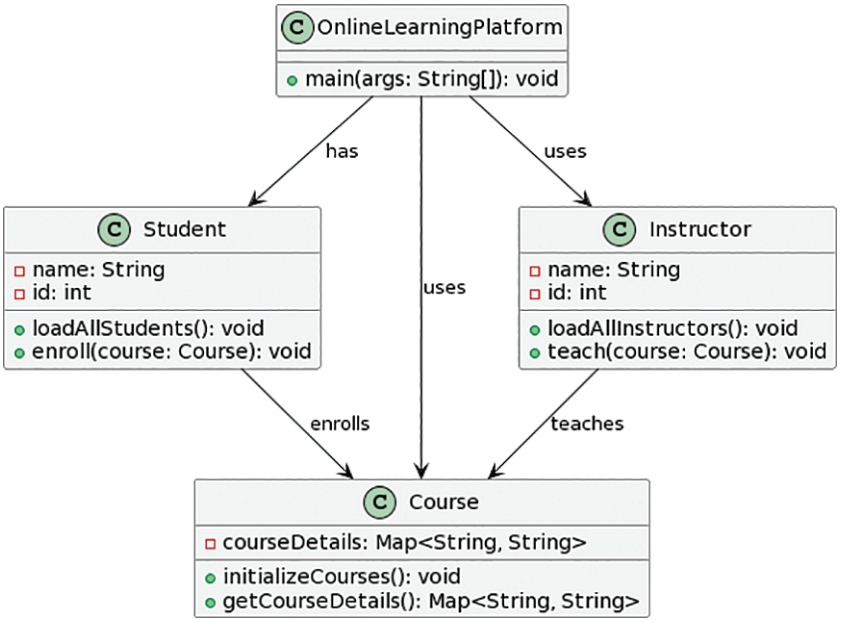

Figure 8.1 Class Diagram for the OnlineLearningPlatform Web Service

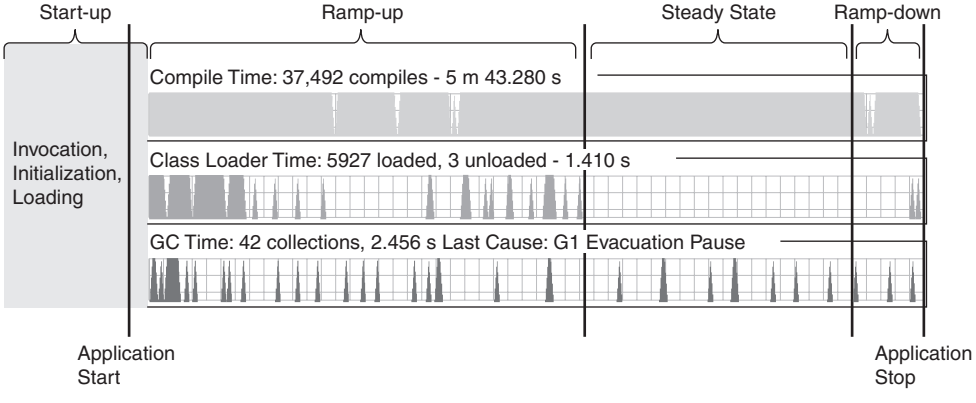

Figure 8.2 Application-Phase Chart Showing Garbage Collection, Compilation, and Class-Loading Timelines

An Application's Life Cycle

For most well-designed applications, we will see a life cycle that goes though the following phases (as shown in Figure 8.3):

- **Start-up:** The JVM is invoked and starts transitioning to the next phase after the application initializes its state. The start-up phase includes JVM initialization, class loading, class initialization, and application initialization work.
- **Ramp-up:** This phase is characterized by a gradual improvement in code generation, leading to increases in the application's performance until it reaches peak performance.
- **Steady-state:** In this phase, the application executes its main workload at peak performance.
- **Ramp-down:** In this phase, the application begins to shut down and release resources.
- **Application stop:** The JVM terminates, and the application stops.

The timeline in Figure 8.3 represents the progression of time from left to right. The application starts after the application completes initialization and then begins to ramp up. The application reaches its steady-state after the ramp-up phase. When the application is ready to stop, it enters the ramp-down phase, and finally the application stops.

Managing State at Start-up and Ramp-up

State management is a pivotal aspect of Java application performance, especially during the start-up and ramp-up phases. The state of an application encompasses various elements, including data, sockets, files, and more. Properly managing this state ensures that the application will run efficiently and lays the foundation for its subsequent operations.

State During Start-up

The state at start-up is a comprehensive snapshot of all the variables, objects, and resources with which the application is interacting at that moment:

- **Static variables:** Variables that retain their values even after the method that created them has finished executing.
- **Heap objects:** Objects that are dynamically allocated during runtime and that reside in a heap region of memory.

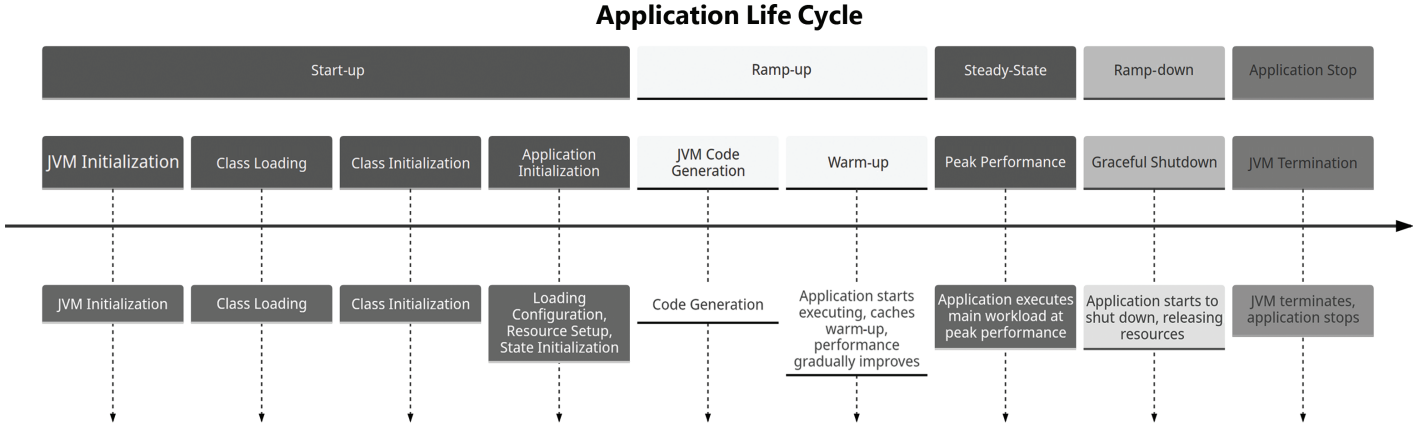

Figure 8.3 An Application Life Cycle

- **Running threads:** The sequences of programmed instructions that are executed by the application.
- **Resources:** File descriptors, sockets, and other resources that the application uses.

The flowchart in Figure 8.4 summarizes the initialization of the application's state, from loading of the static variables to the allocation of resources, and the transition through ramp-up to the steady-state.

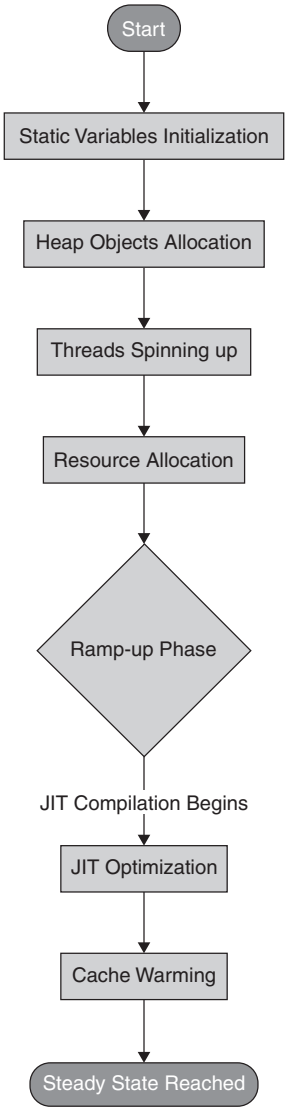

Figure 8.4 Application State Management Flowchart

Transition to Ramp-up and Steady State

Following the start-up phase, the application enters the ramp-up phase, where it starts processing real-world tasks. During this phase, the JIT compiler refines the code based on actual usage patterns. The culmination of the ramp-up phase is marked by the application achieving a steady-state. At this point, performance stabilizes, and the JIT compiler has optimized the majority of the application's code.

As shown in Figure 8.5, the duration from start-up to the steady-state is termed the "time to steady-state." This metric is crucial for gauging the performance of applications, especially those with short lifespans or those operating in serverless environments. In such scenarios, swift start-ups and efficient workload ramp-ups are paramount.

Together, Figure 8.4 and Figure 8.5 provide a visual representation of the application's life cycle, from the start-up phase through to achieving a steady-state, highlighting the importance of efficient state management.

Benefits of Efficient State Management

Proper state management during the start-up and ramp-up phase can reduce the time to steady-state, resulting in the following benefits:

- **Improved performance:** Spending less time on initialization and loading data structures expedites the application's transition to the steady state, thereby enhancing performance.
- **Optimized memory usage:** Efficient state management reduces the amount of memory required to store the application's state, leading to a smaller memory footprint.
- **Better resource utilization:** At the hardware and operating system (OS) levels, streamlined state management can lead to better utilization of the CPU and memory resources and reduce the load on the OS's file and network systems.
- **Containerized environments:** In environments where applications run in containers, state management is even more critical. In these scenarios, resources are often more constrained, and the overhead of the container runtime needs to be accomodated.

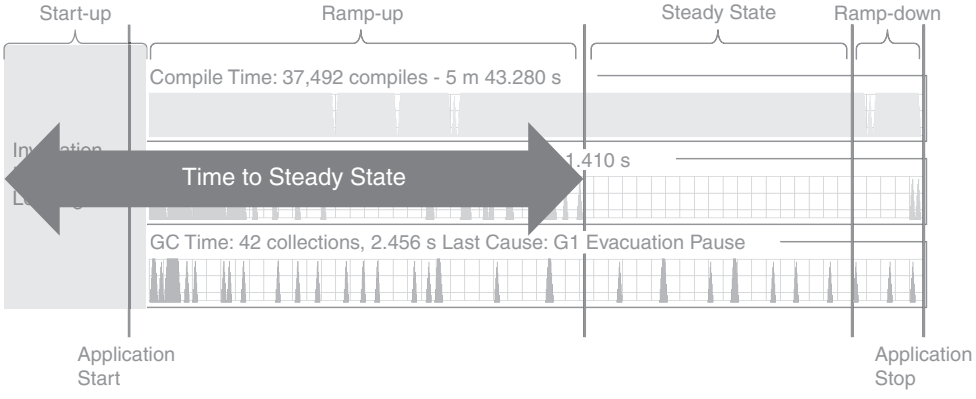

Figure 8.5 An Application Timeline Depicting Time to Steady State

In the current HotSpot VM landscape, one technique stands out for its ability to manage state and improve time-to-steady-state performance: Class Data Sharing.

Class Data Sharing

Class Data Sharing (CDS), a powerful JVM feature, is designed to significantly enhance the start-up performance of Java applications. It accomplishes this feat by preprocessing class files and converting them into an internal format that can be shared among multiple JVM instances. This feature is particularly beneficial for extensive applications that utilize a multitude of classes, as it reduces the time taken to load these classes, thereby accelerating start-up times.

The Anatomy of the Shared Archive File

The shared archive file created by CDS is an efficient and organized data structure that contains several *shared spaces*. Each of these spaces is dedicated to storing a different type of data. For instance,

- ***MiscData* space:** This space is reserved for various metadata types.
- ***ReadWrite* and *ReadOnly* spaces:** These spaces are allocated for the actual class data.

Such a structured approach optimizes data retrieval and streamlines the loading process, contributing to faster start-up times.

Memory Mapping and Direct Usage

Upon JVM initiation, the shared archive file is memory-mapped into its process space. Given that the class data within the shared archive is already in the JVM's internal format, it's directly usable, eliminating the need for any conversions. This direct access further trims the start-up time.

Multi-instance Benefits

The benefits of CDS extend beyond a single JVM instance. The shared archive file can be memory-mapped into multiple JVM processes running on the same machine. This mechanism of memory mapping and sharing is particularly beneficial in cloud environments, where resource constraints and cost-effectiveness are primary concerns. Allowing multiple JVM instances to share the same physical memory pages reduces the overall memory footprint, leading to enhanced resource utilization and cost savings. Furthermore, this sharing mechanism improves the start-up time of subsequent JVM instances, as the shared archive file has already been loaded into memory by the first JVM instance. This proves invaluable in serverless environments where rapid function start-ups in response to events are essential.

Dynamic Dumping with -XX:ArchiveClassesAtExit

The `-XX:ArchiveClassesAtExit` option allows the JVM to dynamically dump the classes loaded by the application to a shared archive file. By preloading the classes that are actually used by the application during the next JVM start, this feature fine-tunes start-up performance.

Using CDS involves the following steps:

1. Generate a shared archive file that contains the preprocessed class data. This is done using the `-Xshare:dump` JVM option.
2. Once the archive file is generated, it can be used by JVM instances using the `-Xshare:on` and `-XX:SharedArchiveFile` options.

For example:

```
java -Xshare:dump -XX:SharedArchiveFile=my_app_cds.jsa -cp my_app.jar
java -Xshare:on -XX:SharedArchiveFile=my_app_cds.jsa -cp my_app.jar MyMainClass
```

In this example, the first command generates a shared archive file for the classes used by `my_app.jar`, and the second command runs the application using the shared archive file.

Ahead-of-Time Compilation

Ahead-of-time (AOT) compilation is a technique that plays a significant role in managing state and improving the time-to-peak performance in the managed runtimes. Introduced in JDK 9, AOT aimed to enhance start-up performance by compiling methods to native code before the JVM was invoked. This precompilation meant that the native code was ready beforehand and eliminated the need to wait for the interpretation of the bytecode. This approach is particularly beneficial for short-lived applications where the JIT compiler might not have enough time to effectively optimize the code.

NOTE Before we get into the intricacies of AOT, it's crucial to understand its counterpart, JIT compilation. In the HotSpot VM, the interpreter is the first to execute, but it doesn't optimize the code. The JIT-ing process brings in the optimizations, but it requires time to gather profiling data for frequently executed methods and loops. As a result, there's a delay, especially for code that needs an optimized state for its efficient execution. Several components contribute to transitioning the code to its optimized state. For instance, the (segmented) code cache maintains the JIT code in its various states (for example, profiled and non-profiled). Additionally, compilation threads manage optimization and deoptimization states via adaptive and speculative optimizations. Notably, the HotSpot VM speculatively optimizes dynamic states, effectively converting them into static states.[1] These elements collectively support the transition of code from its initial interpreted state to its optimized form, which can be viewed as the overhead of JIT compilation.

[1] John Rose pointed out this behavior in his presentation at JVMLS 2023: https://cr.openjdk.org/~jrose/pres/202308-Leyden-JVMLS.pdf.

AOT introduced static compilation to dynamic languages. Prior to HotSpot VM, platforms like IBM's J9 and Excelsior JET had successfully employed AOT. Execution engines that rely on profile-guided techniques to provide adaptive optimizations can potentially suffer from slow start-up times due to the need for a warm-up phase. In HotSpot VM's case, the compilation always begins in interpreted mode. AOT compilation, with its precompiled methods and shared libraries, is a great solution to these warm-up challenges. With AOT at the forefront and the adaptive optimizer in the background, the application can achieve its peak performance in an expedited fashion.

In JDK 9, the AOT feature had experimental status and was built on top of the Graal compiler. Graal is a dynamic compiler that is written in Java, whereas HotSpot VM is written in C++. To support Graal, HotSpot VM needed to support a new JVM compiler interface, called JVMCI.[2] A first pass at this effort was completed for a few known libraries such as `java.base`, `jdk.compiler`, `jdk.vm.compiler`, and a few others.

The AOT code added to JDK 9[3] worked at two levels—tiered and non-tiered, where non-tiered was the default mode of execution and was geared toward increasing application responsiveness. The tiered mode was akin to the T2 level of HotSpot VM's tiered compilation, which is the client compiler with profiling information. After that, the methods that crossed the AOT invocation thresholds were recompiled by the client compiler at the T3 level and full profiling was carried out on those methods, which eventually led to recompilation by the server compiler at the T4 level.

AOT compilation could be used in conjunction with CDS to further improve start-up performance. The CDS feature allows the JVM to share common class metadata across different Java processes, while AOT compilation allowed the JVM to use precompiled code instead of interpreting bytecode. When used together, these features could significantly reduce start-up time and memory footprint.

However, AOT compilers (within the context of the OpenJDK HotSpot VM) faced certain limitations. For instance, AOT-compiled code had to be stored in shared libraries, necessitating the use of position-independent code (PIC).[4] This requirement was due to the fact that the location where the code would be loaded into memory could not be predetermined, preventing the AOT compiler from making location-based optimizations. In contrast, the JIT compiler is allowed to make more assumptions about the execution environment and optimizes the code accordingly.

The JIT compiler can directly reference symbols, eliminating the need for indirections and resulting in more-efficient code. However, these optimizations by the JIT compiler increase start-up time, as the compilation process happens during application execution. By comparison, an interpreter, which executes bytecode directly, doesn't need to spend time compiling the code, but the execution speed of interpreted code is generally much slower than that of compiled code.

---

[2] https://openjdk.org/jeps/243

[3] https://openjdk.org/jeps/295

[4] https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html

NOTE It is important to note that different virtual machines, such as GraalVM, implement AOT compilation in ways that can overcome some of these limitations. In particular, GraalVM's AOT compilation enhanced with profile-guided optimizations (PGO) can achieve peak performance comparable to JIT[5] and even allows for certain optimizations that take advantage of a closed-world assumption, which are not possible in a JIT setting. Additionally, GraalVM's AOT compiler can compile 100% of the code—which could be an advantage over JIT, where some "cold" code may remain interpreted.

Project Leyden: A New Dawn for Java Performance

While AOT promised performance gains, the complexity it added to the HotSpot VM overshadowed its benefits. The challenges of maintaining a separate AOT compiler, the additional maintenance burden due to its reliance on the Graal compiler, and the inefficiencies of generating PIC made it clear that a different approach was needed.[6] This led to the inception of Project Leyden.

The essence of Java application performance lies in understanding and managing the various states particularly through capturing snapshots of variables, objects, and resources, and ensuring they're efficiently utilized. This is where Project Leyden steps in. Introduced in JDK 17, Leyden aims to address the challenges of Java's slow start-up time, slow time-to-peak performance, and large footprint. It's not just about faster start-ups; it's about ensuring that the entire life cycle of a Java application, from start-up to steady-state, is optimized.

Training Run: Capturing Application Behavior for Optimal Performance

Project Leyden introduces the concept of a "training run," which is similar to the step in which we generate the archive of preprocessed classes for CDS. In a training run, we aim to capture all of the application's behavior, especially during the start-up and warm-up phases. This recorded behavior, encompassing method calls, resource allocations, and other runtime intricacies, is then harnessed to generate a custom runtime image. Tailored to the application's specific needs, this image ensures swifter start-ups and a quicker ascent to peak performance. It's Leyden's way of ensuring that Java applications are not just performing adequately, but optimally.

Leyden's precompiled runtime images mark a shift towards enhanced predictability in performance, which is particularly beneficial for applications where traditional JIT compilation might not fully optimize the code due to shorter runtimes. This advancement represents a leap in Java's evolution, streamlining the start-up process and enabling consistent performance.

---

[5] Alina Yurenko. "GraalVM for JDK 21 Is Here!" https://medium.com/graalvm/graalvm-for-jdk-21-is-here-ee01177dd12d#0df7.

[6] https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/

Condensers: Bridging the Gap Between Code and Performance

Whereas CDS serves as the underlying backbone when managing states, Leyden introduces *condensers*—tools akin to the guardians of state management. Condensers ensure that computation, whether directly expressed by a program or done on its behalf, is shifted to the most optimal point in time. By doing so, they preserve the essence of the program while offering developers the flexibility to choose performance over certain functionalities.

Shifting Computation with Project Leyden: A Symphony of States

Java performance optimization relies on the meticulous management of states and their seamless transition from one phase to another. Leyden's approach to shifting computation is a testament to this philosophy. It's not just about moving work around; it's about ensuring that every piece of computation, every variable, and every resource is optimally placed and utilized. This is the art of driving both start-up and warm-up times into the background, making them almost negligible.

Consider our `OnlineLearningPlatform` web service. Prior to Leyden optimization, we could have something like this code snippet:

```
public class OnlineLearningPlatform {
    public static void main(String[] args) {
        // Initialization
        System.out.println("Initializing OnlineLearningPlatform...");

        // Load Student and Course classes
        Student.loadAllStudents();
        Course.initializeCourses();

        // Ramp-up phase: Load Instructor class after its up-to-date list
        Instructor.loadAllInstructors();
        Instructor.teach(new Course()); // This method is a candidate for JIT optimization
    }
}

public class Student {
    private String name;
    private int id;

    public static void loadAllStudents() {
        // Load all students
    }

    public void enroll(Course course) {
        // Enroll student in a course
    }
}
```

```
public class Course {
    private static Map<String, String> courseDetails;

    static {
        courseDetails = new HashMap<>();
        courseDetails.put("Math101", "Basic Mathematics");
        courseDetails.put("Eng101", "English Literature");
        // Additional courses
    }

    public static void initializeCourses() {
        // Initialize courses
    }

    public Map<String, String> getCourseDetails() {
        return courseDetails;
    }
}

public class Instructor {
    private String name;
    private int id;

    public static void loadAllInstructors() {
        // Load all instructors
    }

    public void teach(Course course) {
        // Teach a course
    }
}
```

In this setup, the initialization of `OnlineLearningPlatform`, the loading of the `Student` and `Course` classes, and the ramp-up phase with the `Instructor` class all happen sequentially, potentially leading to longer start-up times. However, with Leyden, we can have something like the following:

```
public class OnlineLearningPlatform {
    public static void main(String[] args) {
        // Initialization with Leyden's shifted computation
        System.out.println("Initializing OnlineLearningPlatform with Leyden optimizations...");

        // Load Student and Course classes from preprocessed images
        Student.loadAllStudentsFromImage();
        Course.initializeCoursesFromImage();
```

```
        // Ramp-up phase: Load Instructor class from preprocessed images and JIT optimizations
        Instructor.loadAllInstructorsFromImage();
        Instructor.teachOptimized();
    }
}

public class Student {
    private String name;
    private int id;

    public static void loadAllStudentsFromImage() {
        // Load all students from preprocessed image
    }

    public void enroll(Course course) {
        // Enroll student in a course
    }
}

public class Course {
    private static Map<String, String> courseDetails;

    static {
        // Load course details from preprocessed image
        courseDetails = ImageLoader.loadCourseDetails();
    }

    public static Map<String, String> initializeCoursesFromImage() {
        // Load course details from preprocessed image and return
        return courseDetails;
    }

    public Map<String, String> getCourseDetails() {
        return courseDetails;
    }
}

public class Instructor {
    private String name;
    private int id;

    public static void loadAllInstructorsFromImage() {
        // Load all instructors from preprocessed image
    }
```

```
    public void teachOptimized(Course course) {
        // Teach a course with JIT optimizations
    }

}
```

The notable shifts are as follows:

- Classes are now sourced from preprocessed images, evident from methods like `loadAllStudentsFromImage()`, `initializeCoursesFromImage()`, and `loadAllInstructorsFromImage()`.
- The static initializer block in the `Course` class now loads course details from a preprocessed image using the `ImageLoader.loadCourseDetails()` method.
- The `Instructor` class introduces `teachOptimized()`, which represents the JIT-optimized teaching method.

By harnessing Leyden's enhancements, the `OnlineLearningPlatform` can leverage preprocessed images for loading classes and JIT-optimized methods. The revised class diagram for our example post-Project Leyden is shown in Figure 8.6.

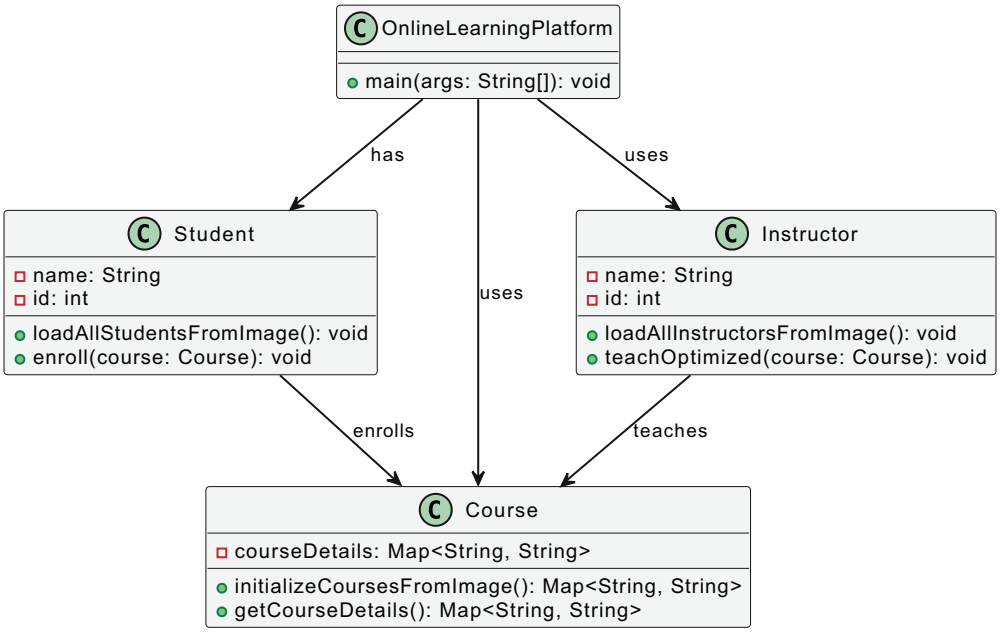

Figure 8.6 New Class Diagram: Post–Project Leyden Optimizations

Looking ahead, Leyden promises a harmonious blend of performance and adaptability. It's about giving developers the tools and the freedom to choose how they want to manage states, how they want to optimize performance, and how they want their applications to interact with the underlying JVM. From introducing condensers to launching the concept of a training run, Leyden is setting the stage for a future in which Java performance optimization is not just a technical endeavor but an art.

As the Java ecosystem continues to evolve, GraalVM emerges as a pivotal innovation, complementing the extension of performance optimization beyond Java's traditional boundaries.

GraalVM: Revolutionizing Java's Time to Steady State

Standing at the forefront of dynamic evolution, GraalVM focuses on optimizing start-up and warm-up performance by leveraging its capabilities and its proficiency in dynamic compilation. It harnesses the power of static images to enhance performance across a wide array of applications.

GraalVM is a polyglot virtual machine that seamlessly supports JVM languages such as Java, Scala, and Kotlin, as well as non-JVM languages such as Ruby, Python, and WebAssembly. Its true prowess lies in its ability to drastically enhance both start-up and warm-up times, ensuring applications not only initiate swiftly but also reach their optimal performance in a reduced time frame.

The heart of GraalVM is the Graal compiler (Figure 8.7). This dynamic compiler, tailored for dynamic or "open-world" execution, similar to the OpenJDK HotSpot VM, is adept at producing C2-grade machine code for a variety of processors. It employs "self-optimizing" techniques, leveraging dynamic and speculative optimizations to make informed predictions about program behavior. If a speculation misses the mark, the compiler can de-optimize and recompile, ensuring applications warm up rapidly and achieve peak performance sooner.

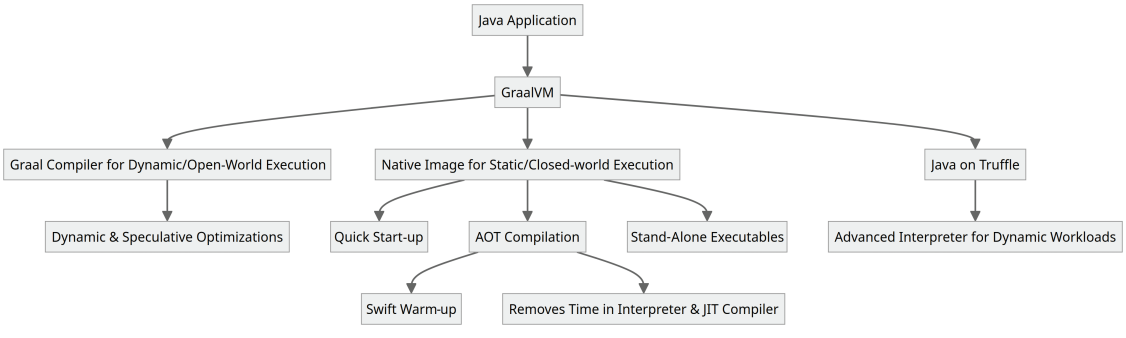

Figure 8.7 GraalVM Optimization Strategies

Beyond dynamic execution, GraalVM offers the power of native image generation for "closed world" optimization. In this scenario, the entire application landscape is known at compile

time. Thus, GraalVM can produce stand-alone executables that encapsulate the application, essential libraries, and a minimal runtime. This approach virtually eliminates time spent in the JVM's interpreter and JIT compiler, ushering in instantaneous start-up and swift warm-up times—a paradigm shift for short-lived applications and microservices.

Simplifying Native Image Generation

GraalVM's AOT compilation amplifies its performance credentials. By precompiling Java bytecode to native code, it minimizes the overhead of the JVM's interpreter during start-up. Furthermore, by sidestepping the traditional JIT compilation phase, applications experience a swifter warm-up, reaching peak performance in a shorter span of time. This is invaluable for architectures like microservices and serverless environments, for which both rapid start-up and quick warm-up are of the essence.

In recent versions of GraalVM, the process of creating native images has been simplified as the native image capability is now included within the GraalVM distribution itself.[7] This enhancement streamlines the process for developers:

```
# Install GraalVM (replace VERSION and OS with your GraalVM version and operating system)
$ curl -LJ https://github.com/graalvm/graalvm-ce-builds/releases/download/vm-VERSION/graalvm-ce-java11-VERSION-OS.tar.gz -o graalvm.tar.gz
$ tar -xzf graalvm.tar.gz

# Set the PATH to include the GraalVM bin directory
$ export PATH=$PWD/graalvm-ce-java11-VERSION-OS/bin:$PATH

# Compile a Java application to a native image
$ native-image -jar your-app.jar
```

Executing this command creates a precompiled, stand-alone executable of the Java application, which includes a minimal JVM and all necessary dependencies. The application can start executing immediately, without needing to wait for the JVM to start up or classes to be loaded and initialized.

Enhancing Java Performance with OpenJDK Support

The OpenJDK community has been working on improving the support for *native-image* building, making it easier for developers to create native images of their Java applications. This includes improvements in areas such as class initialization, heap serialization, and service binding, which contribute to faster start-up times and smaller memory footprints. By providing the ability to compile Java applications to native executables, GraalVM makes Java a more attractive option for high-demand computing environments.

[7] https://github.com/oracle/graal/pull/5995

Introducing Java on Truffle

To provide a complete picture of the execution modes supported by GraalVM, it's important to mention Java on Truffle.[8] This is an advanced Java interpreter that runs atop the Truffle framework and can be enabled using the `-truffle` flag when executing Java programs:

```
# Execute Java with the Truffle interpreter
$ java -truffle [options] class
```

This feature complements the existing JIT and AOT compilation modes, offering developers an additional method for executing Java programs that can be particularly beneficial for dynamic workloads.

Emerging Technologies: CRIU and Project CRaC for Checkpoint/Restore Functionality

OpenJDK continues to innovate via emerging projects such as CRIU and CraC. Together, they contribute to reducing the time to steady-state, expanding the realm of high-performance Java applications.

CRIU (Checkpoint/Restore in Userspace) is a pioneering Linux tool; it was initially crafted by Virtuozzo[9] and subsequently made available as an open-source program. CRIU's designed to support the live migration feature of OpenVZ, a server virtualization solution.[10] The brilliance of CRIU lies in its ability to momentarily freeze an active application, creating a checkpoint stored as files on a hard drive. This checkpoint can later be utilized to resurrect and execute the application from its frozen state, eliminating the need for any alterations or specific configurations.

The following command sequence creates a checkpoint of the process with the specified PID and stores it in the specified directory. The process can then be restored from this checkpoint.

```
# Install CRIU
$ sudo apt-get install criu

# Checkpoint a running process
$ sudo criu dump -t [PID] -D /checkpoint/directory -v4 -o dump.log

# Restore a checkpointed process
$ sudo criu restore -D /checkpoint/directory -v4 -o restore.log
```

Recognizing the potential of CRIU, Red Hat introduced it as a stand-alone project in OpenJDK. The aim was to provide a way to freeze the state of a running Java application, store it, and then restore it later or on a different system. This capability could be used to migrate running applications between systems, save and restore complex applications' state for debugging purposes, and create snapshots of applications for later analysis.

[8] www.graalvm.org/latest/reference-manual/java-on-truffle/

[9] https://criu.org/Main_Page

[10] https://wiki.openvz.org/Main_Page

Building on the capabilities of CRIU, Project CRaC (Coordinated Restore at Checkpoint)—a new project in the Java ecosystem—aims to integrate CRIU's checkpoint/restore functionality into the JVM. This could potentially allow Java applications to be checkpointed and restored, providing another avenue for improving time to steady-state. CRaC is currently in its early stages but represents a promising direction for the future of Java start-up performance optimization.

Let's simulate the process of checkpointing and restoring the state of the `OnlineLearningPlatform` web service:

```
import org.crac.Context;
import org.crac.Core;
import org.crac.Resource;

public class OnlineLearningPlatform implements Resource {
    public static void main(String[] args) throws Exception {
        // Register the platform in a global context for CRaC
        Core.getGlobalContext().register(new OnlineLearningPlatform());

        // Initialization
        System.out.println("Initializing OnlineLearningPlatform...");

        // Load Student and Course classes
        Student.loadAllStudents();
        Course.initializeCourses();

        // Ramp-up phase: Load Instructor class after its up-to-date list
        Instructor.loadAllInstructors();
        Instructor.teach(new Course()); // This method is a candidate for JIT optimization
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        System.out.println("Preparing to checkpoint OnlineLearningPlatform...");
    }

    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
        System.out.println("Restoring OnlineLearningPlatform state...");
    }
}
```

In this enhanced version, the `OnlineLearningPlatform` class implements the `Resource` interface from CRaC. This allows us to define the behavior before a checkpoint (`beforeCheckpoint` method) and after a restore (`afterRestore` method). The platform is registered with CRaC's global context, enabling it to be checkpointed and restored.

The integration of CRIU into the JVM is a complex task because it requires changes to the JVM's internal structures and algorithms to support the freezing and restoring of application state. It also requires coordination with the operating system to ensure that the state of the JVM and the application it is running is correctly captured and restored. Despite the challenges posed, the integration of CRIU into the JVM offers remarkable potential benefits.

Figure 8.8 shows a high-level representation of the relationship between a Java application, Project CRaC, and CRIU. The figure includes the following elements:

1. **Java application:** This is the starting point, where the application runs and operates normally.
2. **Project CRaC:**
   - It provides an API for Java applications to initiate checkpoint and restore operations.
   - When a checkpoint is requested, Project CRaC communicates with CRIU to perform the checkpointing process.
   - Similarly, when a restore is requested, Project CRaC communicates with CRIU to restore the application from a previously saved state.
3. **CRIU:**
   - Upon receiving a checkpoint request, CRIU freezes the Java application process and saves its state as image files on the hard drive.
   - When a restore request is made, CRIU uses the saved image files to restore the Java application process to its checkpointed state.
4. **Checkpoint state and restored state:**
   - The "Checkpoint State" represents the state of the Java application at the time of checkpointing.
   - The "Restored State" represents the state of the Java application after it has been restored. It should be identical to the checkpoint state.
5. **Image files:** These are the files created by CRIU during the checkpointing process. They contain the saved state of the Java application.
6. **Restored process:** This represents the Java application process after it has been restored by CRIU.

Although these projects are still in the early stages of development, they represent promising directions for the future of Java start-up performance optimization. The integration of CRIU into the JVM could have a significant impact on the performance of Java applications. By allowing applications to be checkpointed and restored, it could potentially reduce start-up times, improve performance, and enable new use cases for Java applications.

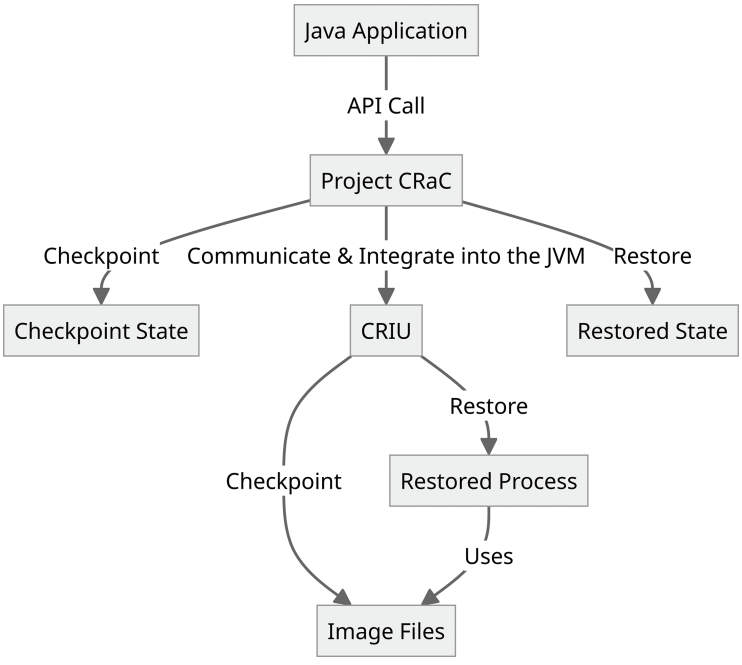

Figure 8.8 Applying Project CRaC and CRIU Strategies to a Java Application

Start-up and Ramp-up Optimization in Serverless and Other Environments

As cloud technologies evolve, serverless and containerized environments have become pivotal in application deployment. Java, with its ongoing advancements, is well suited to addressing the unique challenges of these modern computing paradigms, as shown in Figure 8.9.

- **Serverless cold starts:** GraalVM addresses JVM's cold start delays by pre-compiling applications, providing a quick function invocation response, while CDS complements by reusing class metadata to accelerate start-up times.
- **CDS enhancements:** These techniques are pivotal in both serverless and containerized setups, reusing class metadata for faster start-up and lower memory usage.
- **Prospective optimizations:** Anticipated projects like Leyden and CRaC aim to further refine start-up efficiency and application checkpointing, creating a future in which Java's "cold start" issues become a relic of the past.
- **Containerized dynamics:** Rapid scaling and efficient resource utilization are at the forefront, with Java poised to harness container-specific optimizations for agile performance.

Figure 8.9 encapsulates Java's ongoing journey toward a seamless cloud-native experience, underpinned by robust start-up and ramp-up strategies.

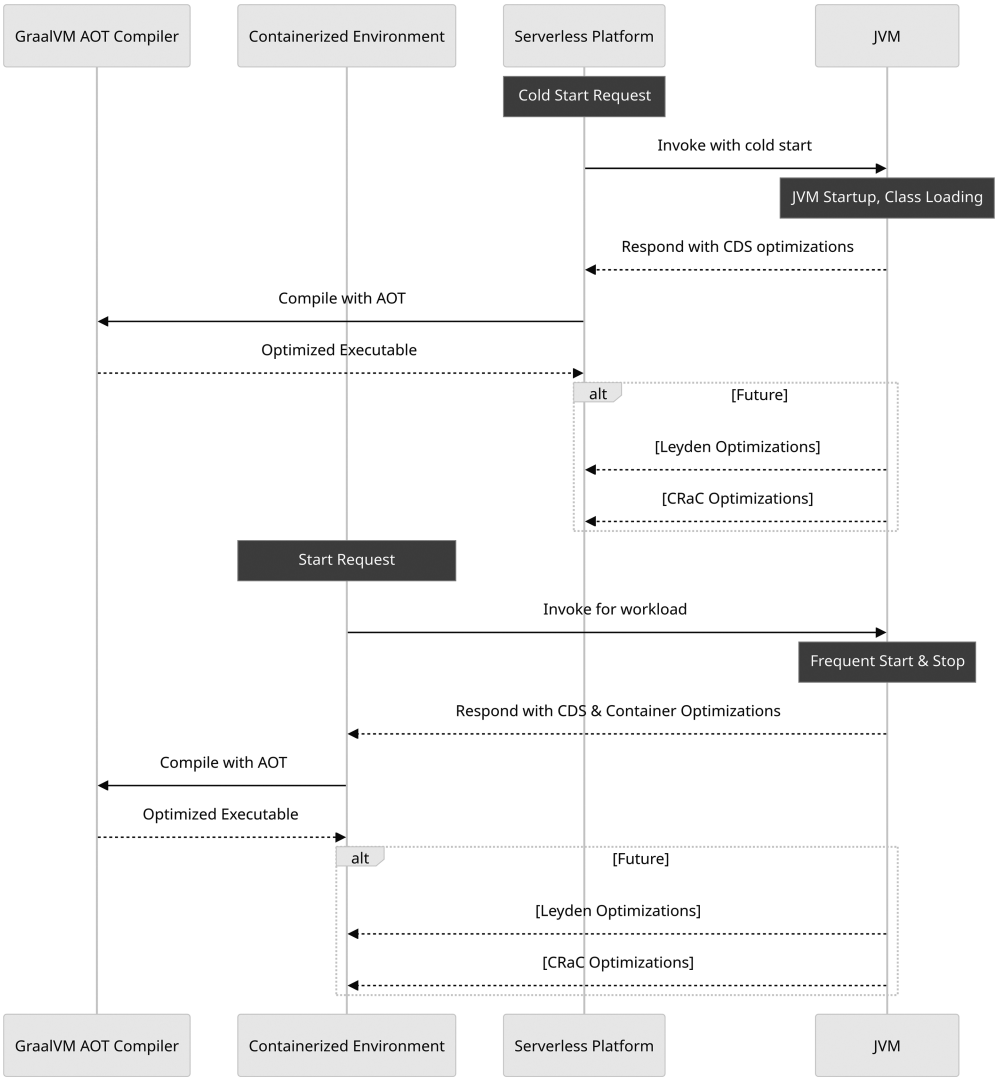

Figure 8.9 JVM Landscape with Containerized and Serverless Platforms

Serverless Computing and JVM Optimization

Serverless computing, with its on-demand execution of application functions, offers a paradigm shift in application deployment and scaling. This approach, which liberates developers from managing servers, enhances operational efficiency and allows for automatic scaling to match workloads. However, serverless environments introduce the "cold start" problem, which is especially pronounced when a dormant function is invoked. This necessitates resource allocation, runtime initiation, and application launch, which can be time-intensive for Java applications due to JVM start-up, class loading, and JIT compilation processes.

The "cold start" problem comes into play in scenarios where a Java application, after being inactive, faces a sudden influx of requests in a serverless environment. This necessitates rapid start-up of multiple application instances, where delays can impact user experience.

To address this issue, JVM optimizations like CDS and JIT compiler enhancements can significantly mitigate cold-start delays, making Java more suitable for serverless computing. Platforms like AWS Lambda and Azure Functions support Java and provide the flexibility to fine-tune JVM configurations. By leveraging CDS and calibrating other JVM options, developers can optimize serverless Java functions, ensuring quicker start-up and ramp-up performance of those functions.

Anticipated Benefits of Emerging Technologies

Emerging technologies like Project Leyden are shaping Java's future performance, particularly for serverless computing. Leyden promises to enhance Java's start-up phase and steady-state efficiency through innovations like "training runs" and "condensers." These developments could substantially cut cold-start times. Even so, it's critical to remember that Leyden's features, as of JDK 21, are still evolving and not ready for production. When Leyden matures, it could transform serverless Java applications, enabling them to quickly react to traffic spikes without the usual start-up delays, thanks to precomputed state storage during "training runs."

Alongside Leyden, Project CRaC offers a hands-on approach. Developers can pinpoint and prepare specific application segments for checkpointing, leading to stored states that can be rapidly reactivated. This approach is particularly valuable in serverless contexts, where reducing cold starts is critical.

These initiatives—Leyden and CRaC—signify a forward leap for Java in serverless environments, driving it toward a future where applications can scale instantaneously, free from the constraints of cold starts.

Containerized Environments: Ensuring Swift Start-ups and Efficient Scaling

In the realm of modern application deployment, containerization has emerged as a game-changer. It encapsulates applications in a lightweight, consistent environment, making them a perfect fit for microservices architectures and cloud-native deployments. Similar to serverless computing, time-to-steady-state performance is crucial. In containerized environments, applications are packaged along with their dependencies into a container, which is then run on a container orchestration platform (for example, Kubernetes). Because containers can be started and stopped frequently based on the workload, having fast start-up and ramp-up times is essential to ensure that the application can quickly scale up to handle the increased load.

Current JVM optimization techniques, such as CDS and JIT compiler enhancements, already contribute significantly to enhancing performance in these contexts. These established strategies include

- **Opt for a minimal base Docker image** to minimize the image size, which consequently improves the start-up and ramp-up durations.
- **Tailor JVM configurations** to better suit the container environment, such as setting the JVM heap size to be empathetic to the memory limits of the container.

- **Strategically layer Docker images** to harness Docker's caching mechanism and speed up the build process.
- **Implement health checks** to ensure the application is running correctly in the container.

By harnessing these techniques, you can ensure that your Java applications in containerized environments start up and ramp up as quickly as possible, thereby providing a better user experience and making more efficient use of system resources.

Figure 8.10 illustrates both the current strategies and the future integration of Leyden's (and CRaC's) methodologies. It's a roadmap that developers can follow today while looking ahead to the benefits that upcoming projects promise to deliver.

GraalVM's Present-Day Contributions

GraalVM is currently addressing some of the challenges that Java faces in modern computing environments. Its notable feature, the native image capability, provides AOT compilation, significantly reducing start-up times—a crucial advantage for serverless computing and microservices architectures. GraalVM's AOT compiler complements the traditional HotSpot VM by targeting specific performance challenges, which is beneficial for scenarios where low latency and efficient resource usage are paramount.

Although GraalVM's AOT compiler is not a one-size-fits-all solution, it offers a specialized approach to Java application performance, catering to the needs of specific use cases in the cloud-native landscape. This versatility positions Java as a strong contender in diverse deployment environments, ensuring that applications remain agile and performant.

Key Takeaways

Java's evolution continues to align with cloud-native requirements. Projects like Leyden and CRaC, alongside the capabilities of GraalVM, showcase Java's adaptability and commitment to performance enhancement and scalability:

- **Adaptable Java evolution:** With initiatives like Project Leyden and the existing strengths of GraalVM, Java demonstrates its adaptability to modern cloud-native deployment paradigms.
- **Optimization techniques:** Understanding and applying JVM optimization techniques is crucial. This includes leveraging CDS and JIT compiler enhancements for better performance.
- **GraalVM AOT compilation in specific scenarios:** For use cases where rapid start-up and a small memory footprint are critical, such as in serverless environments or microservices, GraalVM's AOT compiler offers an effective solution. It complements the traditional HotSpot VM by addressing specific performance challenges of these environments.
- **Anticipating the impact of future projects:** Projects Leyden and CRaC signify Java's forward leap in serverless environments, promising instantaneous scalability and reduced cold-start constraints.

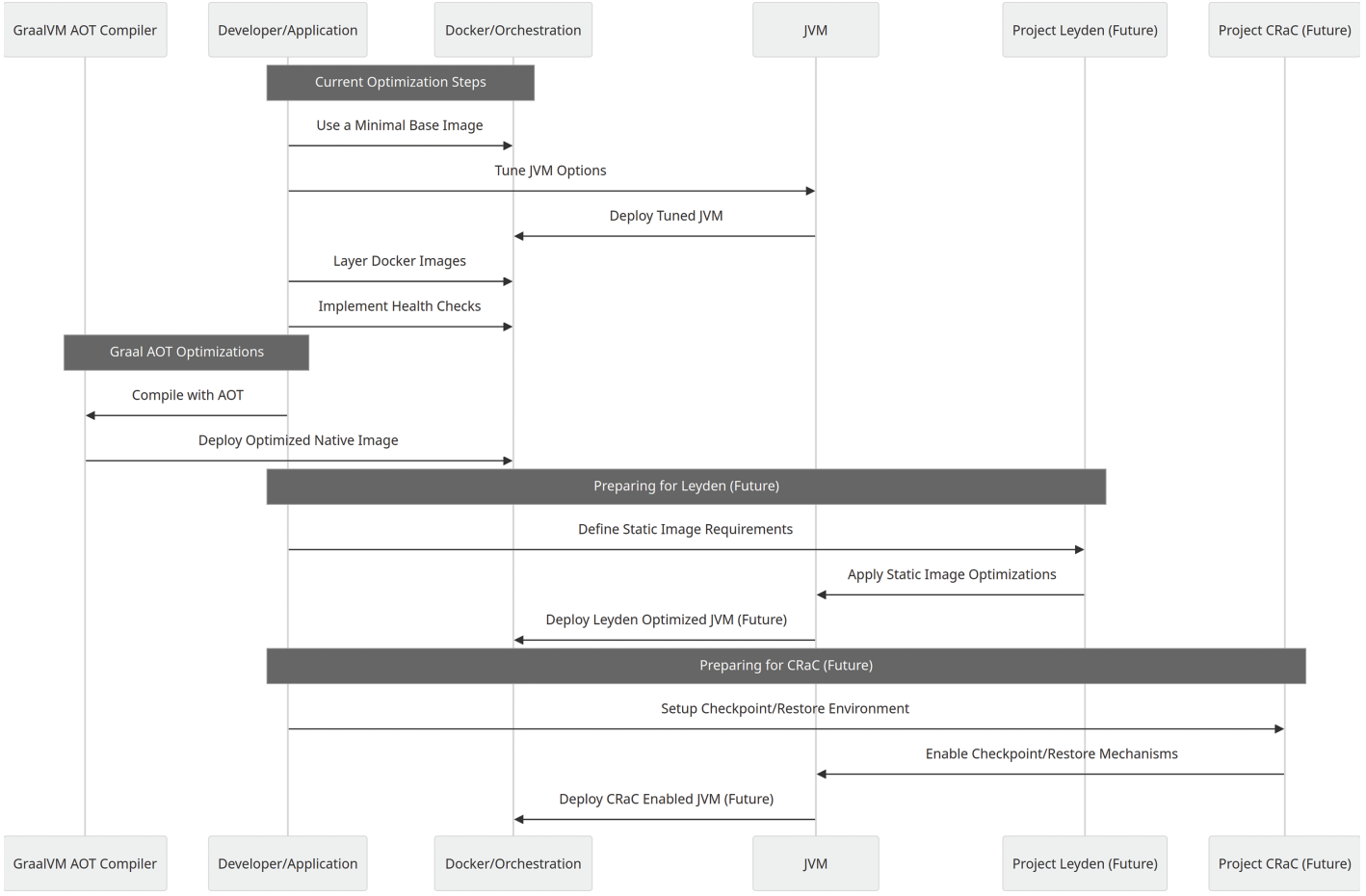

Figure 8.10 Containers' Strategies When Running on the JVM

- **Serverless and containerized performance:** For both serverless and containerized deployments, Java is evolving to ensure swift start-ups and efficient scaling, aligning with the demands of modern cloud applications.
- **Balanced approach:** As the Java ecosystem evolves, a balanced approach in technology adoption is key. Developers should weigh their application needs and the specific benefits offered by each technology, be it current JVM optimizations, GraalVM's capabilities, or upcoming innovations like Leyden and CRaC.

Boosting Warm-up Performance with OpenJDK HotSpot VM

The OpenJDK HotSpot VM is a sophisticated piece of software that employs a multitude of techniques to optimize the performance of Java applications. These techniques are designed to work in harmony, each addressing different aspects of the application life cycle, from start-up to steady-state. This section delves into the intricate workings of the HotSpot VM, shedding light on the mechanisms it uses to boost performance and the roles they play in the broader context of Java application performance.

In the dynamic world of scalable architectures, including microservices and serverless models, Java applications are often subjected to the rigors of frequent restarts and inherently transient lifespans. Such environments amplify the significance of an application's initialization phase, its warm-up period, and its journey to a steady operational state. The efficiency with which an application navigates these phases can make or break its responsiveness, especially when faced with fluctuating traffic demands or the need for rapid scalability. That's where the various optimizations employed by the HotSpot VM—including JIT compilation, adaptive optimization and tiered compilation, and the strategic choice between client and server compilations—come into play.

Compiler Enhancements

In Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine", we looked into the intricacies of HotSpot VM's JIT compiler and its array of compilation techniques (Figure 8.11). As we circle back to this topic, let's summarize the process while emphasizing the optimizations, tailored for start-up, warm-up, and steady-state phases, within the OpenJDK HotSpot VM.

Start-up Optimizations

Start-up in the JVM is primarily handled by the (s)lower tiers. The JVM also takes charge of the *invokedynamic* bootstrap methods (BSM; discussed in detail in Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond") and class initializers.

- **Bytecode generation:** The Java program is first converted into bytecode. This is the initial phase of the Java compilation process, in which the high-level code written by developers is transformed into a platform-agnostic format that can be executed by the JVM (irrespective of the underlying hardware).

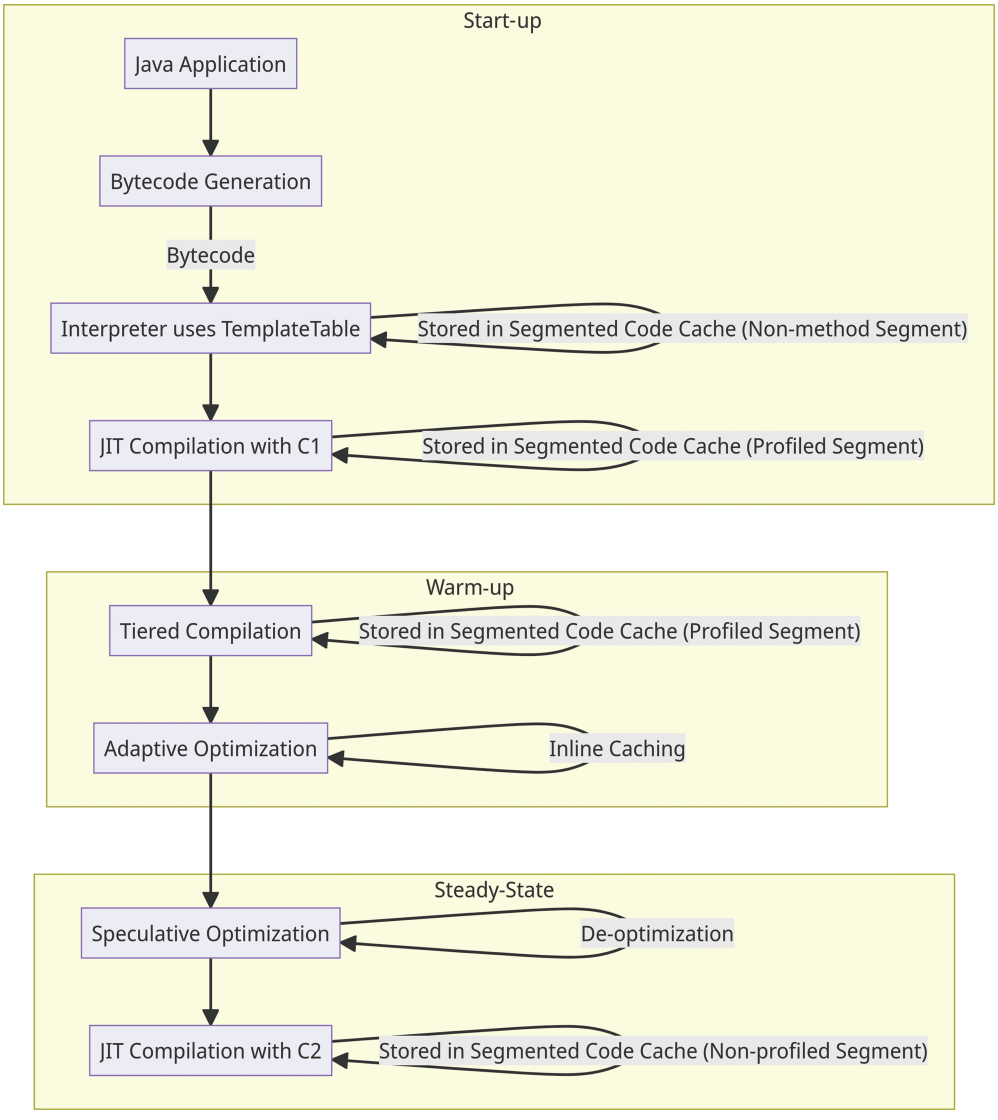

Figure 8.11 OpenJDK Compilation Strategies

- **Interpretation:** The bytecode is then interpreted based on a description table, the *TemplateTable*. This table provides a mapping between bytecode instructions and their corresponding machine code sequences, allowing the JVM to execute the bytecode.
- **JIT compilation with client compiler (C1):** The C1 JIT compiler is designed to provide a balance between quick compilation and a basic level of optimization. It translates the bytecode into native machine code, ensuring that the application quickly moves from the start-up phase and begins its journey toward peak performance. During this phase,

the C1 compiler helps with gathering profiling information about the application behavior. This profiling data is crucial because it informs the subsequent more aggressive optimizations carried out by the C2 compiler. The C1's JIT compilation process is particularly beneficial for methods that are invoked multiple times, because it avoids the overhead of repeatedly interpreting the same bytecode.

- **Segmented *CodeCache*:** The optimized code generated from different tiers of JIT compilation, along with the profiling data, is stored in the *Segmented CodeCache*. This cache is divided into segments, with the *Profiled* segment containing lightly optimized, profiled methods with a short lifetime. It also has a *Non-method* segment that contains non-method code like the bytecode interpreter itself. As we get closer to steady state, the non-profiled, fully optimized methods are also stored in the *CodeCache* in the *Non-profiled* segment. The cache allows for faster execution of frequently used code sequences.

Warm-up Optimizations

As Java applications move beyond the start-up phase, the warm-up phase becomes crucial in setting the stage for peak performance. The JIT compilation, particularly with the client compiler (C1), plays a pivotal role during this transition.

- **Tiered compilation:** A cornerstone of the JVM's performance strategy is tiered compilation, in which code transitions from T0 (interpreted code) to T4 (the highest level of optimization). Tiered compilation allows the JVM to balance the trade-off between faster start-up and peak performance. In the early stages of execution, the JVM uses a faster but less optimized level of compilation to ensure quick start-up. As the application continues to run, the JVM applies more aggressive optimizations to achieve higher peak performance.
- **Adaptive optimization:** The JIT-compiled code undergoes adaptive optimization, which can result in further optimized code or on-stack replacement (OSR). Adaptive optimization involves profiling the running application and optimizing its performance based on its behavior. For example, methods that are called frequently may be further optimized, whereas those superseded may be de-optimized, with their resources reallocated.
- **Inlined caching:** A nuanced technique employed during warm-up is *inlined caching*. With this approach, small yet frequently invoked methods are inlined within the calling method. This strategy minimizes the overhead of method calls and is informed by the profiling data amassed during the warm-up.

Steady-State Optimizations

As Java applications mature in their life cycle, transitioning from warm-up to a steady operational state, the JVM employs a series of advanced optimization techniques. These are designed to maximize the performance of long-running applications, ensuring that they operate at peak efficiency.

- **Speculative optimization:** One of the standout techniques is speculative optimization. Leveraging the rich profiling data accumulated during the warm-up phase, the JVM makes informed predictions about probable execution paths. It then tailors the code optimization based on these anticipations. If any of these educated speculations

proves inaccurate, the JVM is equipped to gracefully revert to a prior, less optimized code version, safeguarding the application's integrity. This strategy shines in enduring applications, where the minor overhead of occasional optimization rollbacks is vastly overshadowed by the performance leaps from accurate speculations.

- **De-optimization:** Going hand in hand with speculative optimization is the concept of de-optimization. This mechanism empowers the JVM to dial back certain JIT compiler optimizations when foundational assumptions are invalidated. A classic scenario is when a newly loaded class overrides a method that had been previously optimized. The ability to revert ensures that the application remains accurate and responsive to dynamic changes.
- **JIT compilation with server compiler (C2):** For long-running applications, the meticulous server compiler (C2) backed with profiling information takes over, applying aggressive and speculative optimizations to performance-critical methods. This process significantly improves the performance of Java applications, particularly for long-running applications where the same methods are invoked multiple times.

Segmented Code Cache and Project Leyden Enhancements

Building upon our detailed discussion of the Segmented *CodeCache* in Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," let's now consider its impact on reducing time-to-steady-state performance of Java applications.

The Segmented *CodeCache* is designed to prioritize the storage of frequently executed code, ensuring that it is readily available for execution. Diving deeper into the mechanics of the *CodeCache*, it's essential to understand the role of the *Code ByteBuffer*. This Buffer acts as an intermediary storage, holding the compiled code before it's moved to the Segmented *CodeCache*. This two-step process ensures efficient memory management and optimized code retrieval. Furthermore, the segmentation of the cache allows for more efficient memory management, as each segment can be sized independently based on its usage. In the context of start-up and warm-up performance, the Segmented *CodeCache* contributes significantly by ensuring that the most critical parts of the application code are compiled and ready for execution as quickly as possible, thereby enhancing the overall performance of Java applications.

With the advent of Project Leyden, this mechanism will see further enhancements. One of the standout features of Project Leyden will be its ability to load the Segmented *CodeCache* directly from an archive. This will bypass the traditional process of tiered code compilation during start-up and warm-up, leading to a significant reduction in start-up times. Instead of compiling the code every time the application starts, Project Leyden will allow the precompiled code to be archived and then directly loaded into the Segmented *CodeCache* upon subsequent start-ups.

This approach will not only accelerate the start-up process but also ensure that the application benefits from optimized code from the get-go. By leveraging this feature, developers will achieve faster application responsiveness, especially in environments where accelerated time-to-steady-state is crucial.

The Evolution from PermGen to Metaspace: A Leap Forward Toward Peak Performance

In the history of Java, prior to the advent of Java 8, the JVM memory included a space known as the Permanent Generation (PermGen). This segment of memory was designated for storing class metadata and statics. However, the PermGen model was not without its flaws. It had a fixed size, which could lead to `java.lang.OutOfMemoryError: PermGen space` errors if the space allocated was inadequate for the demands of the application.

Start-up Implications

The fixed nature of PermGen meant that the JVM had to allocate and deallocate memory during start-up, leading to slower start-up times. In an effort to rectify these shortcomings, Java 8 introduced a significant change: the replacement of PermGen with Metaspace.[11] Metaspace, unlike its predecessor, is not a contiguous heap space, but rather is located in the native memory and used for class metadata storage. It grows automatically by default, and its maximum limit is the amount of available native memory, which is much larger than the typical maximum PermGen size. This crucial modification has contributed to more efficient memory management during start-up, potentially accelerating start-up times.

Warm-up Implications

The dynamic nature of Metaspace, which allows it to grow as needed, ensures that the JVM can quickly adapt to the demands of the application during the warm-up phase. This flexibility reduces the chances of memory-related bottlenecks during this critical phase.

When the Metaspace is filled up, a full garbage collection (GC) is triggered to clear unused class loaders and classes. If GC cannot reclaim enough space, the Metaspace is expanded. This dynamic nature helps avoid the out-of-memory errors that were common with PermGen.

Steady-State Implications

The shift from PermGen to Metaspace ensures that the JVM reaches a steady-state more efficiently. By eliminating the constraints of a fixed memory space for class metadata and statics, the JVM can manage its memory resources and mitigate the risk of out-of-memory errors, resulting in Java applications that are more robust and reliable.

However, although Metaspace can grow as needed, it's not immune to memory leaks. If class loaders are not handled properly, the native memory can fill up, resulting in an `OutOfMemoryError: Metaspace`. Here's how:

1. **Class loader life cycle and potential leaks:**
    a. Each class loader has its own segment of the Metaspace where it loads class metadata. When a class loader is garbage collected, its corresponding Metaspace segment is also freed. However, if live references to the class loader (or to any classes it loaded) persist, the class loader won't be garbage collected, and the Metaspace memory it's using

[11] www.infoq.com/articles/Java-PERMGEN-Removed/

won't be freed. This is how improperly managed class loaders can lead to memory leaks.

   b. Class loader leaks can occur in several other ways. For example, suppose a class loader loads a class that starts a thread, and the thread doesn't stop when the class loader is no longer needed. In that case, the active thread will maintain a reference to the class loader, preventing it from being garbage collected. Similarly, suppose a class loaded by a class loader registers a static hook (for example, a shutdown hook or a JDBC driver), and doesn't deregister the hook when it's done. This can also prevent the class loader from being garbage collected.

   c. In the context of garbage collection, the class loader is a GC root. Any object that is reachable from a GC root is considered alive and is not eligible for garbage collection. So, as long as the class loader remains alive, all the classes it has loaded (and any objects those classes reference) are also considered alive and are not eligible for garbage collection.

2. **`OutOfMemoryError: Metaspace`:** If enough memory leaks accumulate (due to improperly managed class loaders), the Metaspace could fill up. Under that condition, the JVM will trigger a full garbage collection to clear out unused class loaders and classes. If this doesn't free up enough space, the JVM will attempt to expand the Metaspace. If the Metaspace can't be expanded because not enough native memory is available, an `OutOfMemoryError: Metaspace` will be thrown.

It's crucial to monitor Metaspace usage and adjust the maximum size limit as needed. Several JVM options can be used to control the size of the Metaspace:

- **`-XX:MetaspaceSize=[size]`** sets the initial size of the Metaspace. If it is not specified, the Metaspace will be dynamically resized based on the application's demand.
- **`-XX:MaxMetaspaceSize=[size]`** sets the maximum size of the Metaspace. If it is not specified, the Metaspace can grow without limit, up to the amount of available native memory.
- **`-XX:MinMetaspaceFreeRatio=[percentage]` and `-XX:MaxMetaspaceFreeRatio=[percentage]`** control the amount of free space allowed in the Metaspace after GC before it will resize. If the percentage of free space is less than or greater than these thresholds, the Metaspace will shrink or grow accordingly.

Tools such as VisualVM, JConsole, and Java Mission Control can be used to monitor Metaspace usage and provide valuable insights into memory usage, GC activity, and potential memory leaks. These tools can also help identify class loader leaks by showing the number of classes loaded and the total space occupied by these classes.

Java 16 brought a significant upgrade to Metaspace with the introduction of JEP 387: *Elastic Metaspace*.[12] The primary objectives of JEP 387 were threefold:

- Efficiently relinquish unused class-metadata memory back to the OS
- Minimize the overall memory footprint of Metaspace
- Streamline the Metaspace codebase for enhanced maintainability

[12] https://openjdk.org/jeps/387

Here's a breakdown of the key changes that JEP 387 brought to the table:

- **Buddy-based allocation scheme:** This mechanism organizes memory blocks based on size, facilitating rapid allocation and deallocation in Metaspace. It's analogous to a buddy system, ensuring efficient memory management.
- **Lazy memory commitment:** A judicious approach in which the JVM commits memory only when it's imperative. This strategy curtails the JVM's memory overhead, especially when the allocated Metaspace significantly surpasses its actual utilization.
- **Granular memory management:** This enhancement empowers the JVM to manage Metaspace in finer segments, mitigating internal fragmentation and optimizing memory consumption.
- **Revamped reclamation policy:** A proactive strategy enabling the JVM to swiftly return unused Metaspace memory to the OS. This is invaluable for applications with fluctuating Metaspace usage patterns, ensuring a consistent and minimal memory footprint.

These innovations adeptly address Metaspace management challenges, including memory fragmentation. In summary, the dynamic nature of Metaspace, complemented by the innovations introduced in JEP 387, underscores Java's commitment to optimizing memory utilization, reducing fragmentation, and promptly releasing unused memory back to the OS.

Conclusion

In this chapter, we have comprehensively explored JVM start-up and warm-up performance—a critical aspect of Java application performance. Various optimization techniques introduced into the Java ecosystem, such as CDS, GraalVM, and JIT compiler enhancements, have significantly improved start-up and ramp-up times. For microservices architectures, serverless computing, and containerized environments, rapid initializations and small memory footprints are crucial.

As we step into the future, the continuous evolution of Java's performance capabilities remains at the forefront of technological innovation. The introduction of Project Leyden, with its training runs, and the emergence of CRIU and Project CRaC further amplify the potential to drive Java performance optimization.

As developers and performance engineers, it's important that we stay informed about these kinds of advancements and understand how to apply them in different environments. By doing so, we can ensure that our Java applications are as efficient and performant as possible, providing the best possible user experiences.

Chapter 9

Harnessing Exotic Hardware: The Future of JVM Performance Engineering

Introduction to Exotic Hardware and the JVM

In the ever-advancing realm of computational technology, specialized hardware components like graphics processing units (GPUs), field programmable gate arrays (FPGAs), and a plethora of other accelerators are making notable strides. Often termed "exotic hardware," these components are just the tip of the iceberg. Cutting-edge hardware accelerators, including tensor processing units (TPUs), application-specific integrated circuits (ASICs), and innovative AI chips such as Axelera,[1] are reshaping the performance benchmarks across diverse applications. While these powerhouses are primarily tailored to supercharge machine learning and intricate data processing tasks, their prowess isn't limited to just that arena. JVM-based applications, with the aid of specialized interfaces and libraries, can tap into these resources. For example, GPUs, celebrated for their unparalleled parallel computation capabilities, can be a boon for Java applications, turbocharging data-heavy tasks.

Historically, the JVM, as a platform-independent execution environment, has been associated with general-purpose computing in which the bytecode is compiled to machine code for the CPU. This design has allowed the JVM to provide a high degree of portability across different hardware platforms, as the same bytecode can run on any device with a JVM implementation.

However, the rise of exotic hardware has brought new opportunities and challenges for the JVM. Although general-purpose CPUs are versatile and capable, they are often complemented by specialized hardware components that provide capabilities tailored for specific tasks. For instance, while CPUs have specialized instruction sets like AVX512 (Advanced Vector Extensions)[2] for Intel processors and SVE (Scalable Vector Extension)[3] for Arm architectures, which enhance performance for specific tasks, other hardware accelerators offer the ability to

[1] www.axelera.ai/

[2] https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

[3] https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions

perform large-scale matrix operations or specialized computations essential for tasks like deep learning. To take full advantage of these capabilities, the JVM needs to be able to compile bytecode into machine code that can run efficiently on such hardware.

This has led to the development of new language features, APIs, and toolchains designed to bridge the gap between the JVM and specialized hardware. For example, the Vector API, which is part of Project Panama, allows developers to express computations that can be efficiently vectorized on hardware that supports vector operations. By effectively harnessing these hardware capabilities, we can achieve remarkable performance improvements.

Nevertheless, the task of effectively utilizing these specialized hardware components in JVM-based applications is far from straightforward. It necessitates adaptations in both language design and toolchains to tap into these hardware capabilities effectively, including the development of APIs capable of interfacing with such hardware and compilers that can generate code optimized for these components. However, challenges such as managing memory access patterns, understanding hardware-specific behaviors, and dealing with the heterogeneity of hardware platforms can make this a complex task.

Several key concepts and projects have been instrumental in this adaptation process:

- **OpenCL:** An open standard that enables portable parallel programming across heterogeneous systems, including CPUs, GPUs, and other processors. Although OpenCL code can be executed on a variety of hardware platforms, its performance is *not* universally portable. In other words, the efficiency of the same OpenCL code can vary significantly depending on the hardware on which it's executed. For instance, while a commodity GPU might offer substantial performance improvements over sequential code, the same OpenCL code might run more slowly on certain FPGAs.[4]
- **Aparapi:** A Java API designed for expressing data parallel workloads. Aparapi translates Java bytecode into OpenCL, enabling its execution on various hardware accelerators, including GPUs. Its runtime component manages data transfers and the execution of the generated OpenCL code, providing performance benefits for data-parallel tasks. Although Aparapi abstracts much of the complexity, achieving optimal performance on specific hardware might require tuning.
- **TornadoVM:** An extension of the OpenJDK GraalVM, TornadoVM offers the unique benefit of dynamically recompiling and optimizing Java bytecode for different hardware targets at runtime. This allows Java programs to automatically adapt to available hardware resources, such as GPUs and FPGAs, without any code changes. The dynamic nature of TornadoVM ensures that applications can achieve optimal performance portability based on the specific hardware and the nature of the application.
- **Project Panama:** An ongoing project in the OpenJDK community aimed at improving the connection between the JVM's interoperability with native code. It focuses on two main areas:
  - **Vector API:** Designed for vector computations, this API ensures runtime compilation to the most efficient vector hardware instructions on supported CPU architectures.

[4] Michail Papadimitrioua, Juan Fumero, Athanasios Stratikopoulosa, Foivos S. Zakkakb, and Christos Kotselidis. "Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs." *Art, Science, and Engineering of Programming* 5, no. 2 (2020). https://arxiv.org/ftp/arxiv/papers/2010/2010.16304.pdf.

- **Foreign Function and Memory (FFM) API:** This tool allows a program to call routines or utilize services written in a different language. Within the scope of Project Panama, the FFM enables Java code to interact seamlessly with native libraries, thereby enhancing the interoperability of Java with other programming languages and systems.

This chapter explores the challenges associated with JVM performance engineering and discusses some of the solutions that have been proposed. Although our primary focus will be on the OpenCL-based toolchain, its important to acknowledge that CUDA[5] is one of the most widely adopted parallel programming models for GPUs. The significance of language design and toolchains cannot be overstated when it comes to effectively utilizing the capabilities of exotic hardware. To provide a practical understanding, a series of illustrative case studies will be presented, showcasing examples of how these challenges have been addressed and the opportunities that exotic hardware presents for JVM performance engineering.

Exotic Hardware in the Cloud

The availability of cloud computing has made it convenient for end users to gain access to specialized or heterogeneous hardware. In the past, leveraging the power of exotic hardware often required a significant upfront investment in physical hardware. This requirement was a formidable barrier for many developers and organizations, particularly those with limited resources.

In recent years, the cloud computing revolution has dramatically changed this landscape. Now, developers can access and leverage the power of exotic hardware without making a substantial initial investment. This has been made possible by major cloud service providers, including Amazon Web Services (AWS),[6] Google Cloud,[7] Microsoft Azure,[8] and Oracle Cloud Infrastructure.[9] These providers have extended their offerings with virtual machines that come equipped with GPUs and other accelerators, enabling developers to harness the power of exotic hardware in a flexible and cost-effective manner. NVIDIA has been at the forefront of GPU virtualization, offering a flexible approach to video encoding/decoding and broad hypervisor support, but AMD and Intel also have their own distinct methods and capabilities.[10]

To address the virtualization complexities, many cloud "provisioners" ensure that only one host requiring the GPU is deployed on physical hardware, thus gaining full and exclusive access to it. Although this approach allows other (non-GPU) hosts to utilize the CPUs, it does speak to some of the underlying complexities of utilizing exotic hardware in the cloud. Indeed, harnessing specialized hardware components in the cloud presents its own set of challenges, particularly from a software and runtime perspective. Some of these challenges are summarized in the following subsections.

---

[5] https://developer.nvidia.com/cuda-zone

[6] https://aws.amazon.com/ec2/instance-types/f1/

[7] https://cloud.google.com/gpu

[8] www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/

[9] www.oracle.com/cloud/compute/gpu/

[10] Gabe Knuth. "NVIDIA, AMD, and Intel: How They Do Their GPU Virtualization." *TechTarget* (September 26, 2016). www.techtarget.com/searchvirtualdesktop/opinion/NVIDIA-AMD-and-Intel-How-they-do-their-GPU-virtualization.

Hardware Heterogeneity

The landscape of cloud computing is marked by a wide array of hardware offerings, each with unique features and capabilities. For instance, newer Arm devices are equipped with cryptographic accelerators that significantly enhance the speed of cryptographic operations. In contrast, Intel's AVX-512 instructions expand the width of SIMD vector registers to 512 bits, facilitating parallel processing of more data.

Arm's technology spectrum also includes NEON and SVE. Whereas NEON provides SIMD instructions for Arm v7 and Arm v8, SVE allows CPU designers to select the SIMD vector length that best aligns with their requirements, ranging from 128 bits to 2048 bits in 128-bit increments. These variations in data widths and access patterns can significantly influence computational performance.

The diversity also extends to GPUs, FPGAs, and other accelerators, albeit with their availability varying significantly across cloud providers. This hardware heterogeneity necessitates software and runtimes to be adaptable and flexible, capable of optimizing performance based on the specific hardware in use. Adaptability in this context often involves taking different optimization paths depending on the hardware, thereby ensuring efficient utilization of the available resources.

API Compatibility and Hypervisor Constraints

Specialized hardware often requires specific APIs to be used effectively. These APIs provide a way for software to interact with the hardware, allowing for tasks such as memory management and computation to be performed on the hardware. One such widely used API for general-purpose computation on GPUs (GPGPU) is OpenCL. However, JVM, which is designed to be hardware-agnostic, may not natively support these specialized APIs. This necessitates the use of additional libraries or tools, such as the Java bindings for OpenCL,[11] to bridge this gap.

Moreover, the hypervisor used by the cloud provider plays a crucial role in managing and isolating the resources of the virtual machines, providing security and stability. However, it can impose additional constraints on API compatibility. The hypervisor controls the interaction between the guest operating system and the hardware, and while it is designed to support a wide range of operations, it may not support all the features of the specialized APIs. For example, memory management in NVIDIA's CUDA or OpenCL requires direct access to the GPU memory, which may not be fully supported by the hypervisor.

An important aspect to consider is potential security concerns, particularly with regard to GPU memory management. Traditional hypervisors and their handling of the input–output memory management unit (IOMMU) can prevent regular memory from leaking between different hosts. However, the GPU often falls outside this "scope." The GPU driver, which is usually unaware that it is running in a virtualized environment, relies on maintaining memory resident between kernel dispatches. This raises concerns about data isolation—namely, whether data written by one host is truly isolated from another host. With GPGPU, in which the GPU is not just processing images but is also used for general computation tasks, any potential data leakage could be significantly more consequential. Modern cloud infrastructures have made strides in addressing these issues, yet the potential risk still exists, underlining the importance of allowing only one host to have access to the GPU.

[11] www.jocl.org/

Although these constraints might seem daunting, it's worth reiterating the vital role that hypervisors play in maintaining secure and isolated environments. Developers, however, must acknowledge these factors when they're aiming to fully leverage the capabilities of specialized hardware in the cloud.

Performance Trade-offs

The use of virtualized hardware in the cloud is a double-edged sword. On the one hand, it provides flexibility, scalability, and isolation, making it easier to manage and deploy applications. On the other hand, it can introduce performance trade-offs. The overhead of virtualization, which is necessary to provide these benefits, can sometimes offset the performance benefits of the specialized hardware.

This overhead is often attributable to the hypervisor, which needs to translate calls from the guest operating system to the host hardware. Hypervisors are designed to minimize this overhead and are continually improving in efficiency. However, in some scenarios, the burden can still impact the performance of applications running on exotic hardware. For instance, a GPU-accelerated machine learning application might not achieve the expected speed-up due to the overhead of GPU virtualization, particularly if the application hasn't been properly optimized for GPU acceleration or if the virtualization overhead isn't properly managed.

Resource Contention

The shared nature of cloud environments can lead to inconsistent performance due to contention for resources. This is often referred to as the "noisy neighbor" problem, as the activity of other users on the same physical hardware can impact your application's performance. For example, suppose multiple users are running GPU-intensive tasks on the same physical server: They might experience reduced performance due to contention for GPU resources if a noisy neighbor can monopolize GPU resources, causing other users' tasks to run more slowly. This is a common issue in cloud computing infrastructure, where resources are shared among multiple users. To mitigate this problem, cloud providers often implement resource allocation strategies to ensure fair usage of resources among all users. However, these strategies are not always perfect and can lead to performance inconsistencies due to resource contention.

Cloud-Specific Limitations

Cloud environments can impose additional limitations that are not present in on-premises environments. For example, cloud providers typically limit the amount of resources, such as memory or GPU compute units, that a single virtual machine can use. This constraint can limit the performance benefits of exotic hardware in the cloud. Furthermore, the ability to use certain types of exotic hardware may be restricted to specific cloud regions or instance types. For instance, Google Cloud's A2 VMs, which support the NVIDIA A100 GPU, are available only in selected regions.

Both industry and the research community have been actively working on solutions to these challenges. Efforts are being made to standardize APIs and develop hardware-agnostic programming models. As discussed earlier, OpenCL provides a unified, universal standard environment for parallel programming across heterogenous computing systems. It's designed to harness the computational power of diverse hardware components within a single node, making it ideal for efficiently running domain-specific workloads, such as data parallel applications and big data processing. However, to fully leverage OpenCL's capabilities within the JVM, Java bindings for OpenCL are essential. Although OpenCL addresses the computational aspect of this challenge within a node, high-performance computing platforms, especially those used in supercomputers, often require additional libraries like MPI for inter-node communication.

During my exploration of these topics, I had the privilege of discussing them with Dr. Juan Fumero, a leading figure behind the development of TornadoVM, a plug-in for OpenJDK that aims to address these challenges. Dr. Fumero shared a poignant quote from Vicent Natol of HPC Wire: "CUDA is an elegant solution to the problem of representing parallelism in algorithms—not all algorithms, but enough to matter." This sentiment applies not only to CUDA but also to other parallel programming models such as OpenCL, oneAPI, and more. Dr. Fumero's insights into parallel programming and its challenges are particularly relevant when considering solutions like TornadoVM. Developed under his guidance, TornadoVM seeks to harness the power of parallel programming in the JVM ecosystem by allowing Java programs to automatically run on heterogeneous hardware. Designed to take advantage of the GPUs and FPGAs available in cloud environments, TornadoVM accelerates Java applications. By providing a high-level programming model and handling the complexities of hardware heterogeneity, API compatibility, and potential security concerns, TornadoVM makes it easier for developers to leverage the power of exotic hardware in the cloud.

The Role of Language Design and Toolchains

To effectively utilize the capabilities of exotic hardware, both language design and toolchains need to adapt to their new demands. This involves several key considerations:

- **Language abstractions:** The programming language should offer intuitive high-level abstractions, enabling developers to craft code that can execute efficiently on different types of hardware. This involves designing language features that can express parallelism and take advantage of the unique features of exotic hardware. For example, the Vector API in Project Panama provides a way for developers to express computations that can be efficiently vectorized on hardware that supports vector operations.
- **Compiler optimizations:** The toolchain, and in particular the compiler, plays a crucial role in translating high-level language abstractions into efficient low-level code that can run on exotic hardware. This involves developing sophisticated optimization techniques that can take advantage of the unique features of different types of hardware. For example, the TornadoVM compiler can generate OpenCL, CUDA, and SPIR-V code. Furthermore, it produces code optimized for FPGAs, RISC-V with vector instructions, and Apple M1/M2 chips. This allows TornadoVM to be executed on a vast array of computing systems, from IoT devices like NVIDIA Jetsons to PCs, cloud environments, and even the latest consumer-grade processors.

- **Runtime systems:** The runtime system needs to be able to manage and schedule computations on different types of hardware. This involves developing runtime systems that can handle the complexities of heterogeneous hardware, such as managing memory across different types of devices and scheduling computations to minimize data transfer. Consider the task of image processing, in which different filters or transformations are applied to an image. The way these operations are scheduled on a GPU can drastically affect performance. For instance, processing an image in smaller segments or "blocks" might be more efficient for certain filters, whereas other operations might benefit from processing the image as larger contiguous chunks. The choice of how to divide and process the image—whether in smaller blocks or in larger sections—can be analogous to the difference between a filter being applied in real time versus experiencing a noticeable delay. In the runtime system, such computations must be adeptly scheduled to minimize data transfer overheads and to fully harness the hardware's capabilities.
- **Interoperability:** The language and toolchain should provide mechanisms that allow for interoperability with the existing libraries and frameworks that are designed to work with exotic hardware. This can involve providing Foreign Function Interface (FFI) mechanisms, like those in Project Panama, that allow Java code to interoperate with native libraries.
- **Library support:** Although some libraries have been developed to support exotic hardware, these libraries are often specific to certain types of hardware and may not be available or optimized for all types of hardware offered by cloud providers. Specializations in these libraries can include optimizations for specific hardware architectures, which can lead to significant performance differences when running on different types of hardware.

These considerations significantly influence the design and implementation of the projects that aim to better leverage exotic hardware capabilities. For instance, the design of Aparapi, an API for expressing data parallel workloads, has been heavily influenced by the need for language abstractions that can express parallelism and take advantage of the unique features of exotic hardware. Similarly, the development of TornadoVM has been guided by the need for a runtime system that can manage and schedule computations on different types of hardware.

In the case studies that follow, we will delve into these projects in more detail, exploring how they address the challenges of exotic hardware utilization and how they are influenced by the considerations discussed earlier.

Case Studies

It's important to ground our discussion of exotic hardware and the JVM in real-world examples, and what better way to do that than through a series of case studies. Each of these projects—LWJGL, Aparapi, Project Sumatra, CUDA4J (the joint IBM–NVIDIA project), TornadoVM, and Project Panama—offers unique insights into the challenges and opportunities presented by hardware accelerators.

- **LWJGL (Lightweight Java Game Library)**[12] serves as a foundational case, demonstrating the use of the Java Native Interface (JNI) to enable Java applications to interact with

[12] www.lwjgl.org

native APIs for a range of tasks, including graphics and compute operations. It provides a practical example of how existing JVM mechanisms can be used to leverage specialized hardware.

- **Aparapi** showcases how language abstractions can be designed to express parallelism and take advantage of the unique features of heterogeneous hardware. It translates Java bytecode into OpenCL at runtime, enabling the execution of parallel operations on the GPU.
- **Project Sumatra**, although no longer active, was a significant effort to enhance the JVM's ability to offload computations to the GPU by leaning on Java 8's Stream API to express parallelism. The experiences and lessons from Project Sumatra continue to inform current and future projects.
- Running in parallel to Project Sumatra, **CUDA4J** emerged as a joint effort by IBM and NVIDIA.[13] This project leveraged the Java 8 Stream API, enabling Java developers to write GPU computations that the CUDA4J framework translated into CUDA kernels. It demonstrated the power of collaborative efforts in the community to enhance the compatibility between the JVM and exotic hardware, particularly GPUs.
- **TornadoVM** demonstrates how runtime systems can be developed to manage and schedule computations on different types of hardware. It provides a practical solution to the challenges of hardware heterogeneity and API compatibility.
- **Project Panama** provides a glimpse into the future of the JVM. It focuses on improving the connection between the JVM and foreign APIs, including libraries and hardware accelerators, through the introduction of a new FFM API and a Vector API. These developments represent a significant evolution in JVM design, enabling more efficient and streamlined interactions with exotic hardware.

These projects were chosen not only for their technical innovations but also for their relevance to the evolving landscape of JVM and specialized hardware. They represent significant efforts in the community to adapt the JVM to better leverage the capabilities of exotic hardware, and as such, provide valuable insights for our discussion.

LWJGL: A Baseline Example

LWJGL serves as a prime example of how Java interacts with exotic hardware. This mature and widely used library provides Java developers with access to a range of native APIs, including those for graphics (OpenGL, Vulkan), audio (OpenAL), and parallel computation (OpenCL). Developers can use LWJGL in their Java applications to access these native APIs. For instance, a developer might use LWJGL's bindings for OpenGL to render 3D graphics in a game or simulation. This involves writing Java code that calls methods in the LWJGL library, which in turn invoke the corresponding functions in the native OpenGL library.

[13] Despite its niche nature, CUDA4J continues to find application on IBM Power platforms with NVIDIA hardware: www.ibm.com/docs/en/sdk-java-technology/8?topic=only-cuda4j-application-programming-interface-linux-windows.

Here is a simple example of how one might use LWJGL to create an OpenGL context and clear the screen:

```
try (MemoryStack stack = MemoryStack.stackPush()) {
    GLFWErrorCallback.createPrint(System.err).set();
    if (!glfwInit()) {
        throw new IllegalStateException("Unable to initialize GLFW");
    }
    long window = glfwCreateWindow(800, 600, "Shrek: The Musical", NULL, NULL);
    glfwMakeContextCurrent(window);
    GL.createCapabilities();
    while (!glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Set the color to swamp green
        glClearColor(0.13f, 0.54f, 0.13f, 0.0f);
        // Draw a simple 3D scene...
        drawScene();
        // Draw Shrek's house
        drawShrekHouse();
        // Draw Lord Farquaad's castle
        drawFarquaadCastle();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

In this example, we're creating a simple 3D scene for a game based on *Shrek: The Musical*. We start by initializing GLFW and creating a window. We then make the window's context current and create an OpenGL context. In the main loop of our game, we clear the screen, set the color to swamp green (to represent Shrek's swamp), draw our 3D scene, and then draw Shrek's house and Lord Farquaad's castle. We then swap buffers and poll for events. This is a very basic example, but it gives you a sense of how you might use LWJGL to create a 3D game in Java.

LWJGL and the JVM

LWJGL interacts with the JVM primarily through JNI, the standard mechanism for calling native code from Java. When a Java application calls a method in LWJGL, the LWJGL library uses JNI to invoke the corresponding function in the native library (such as `OpenGL.dll` or `OpenAL.dll`). This allows the Java application to leverage the capabilities of the native library, even though the JVM itself does not directly support these capabilities.

Figure 9.1 illustrates the flow of control and data between Java code, LWJGL, and the native libraries. At the top is the application that runs on top of the JVM. This application uses the Java APIs provided by LWJGL to interact with the native APIs. The LWJGL provides a bridge between the JVM and the native APIs, allowing the application to leverage the capabilities of the native hardware.

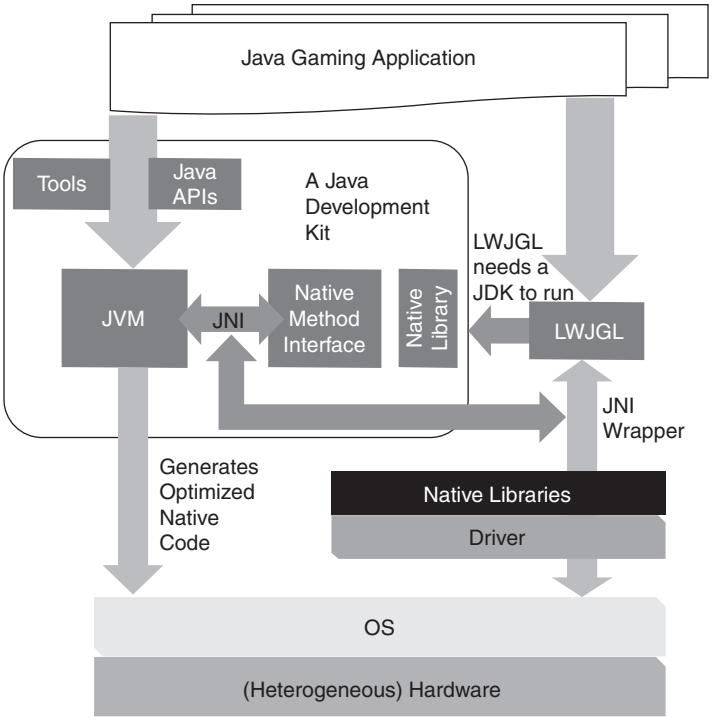

Figure 9.1 LWJGL and the JVM

The LWJGL uses JNI to call native libraries—represented by the JNI Wrapper in Figure 9.1. The JNI Wrapper is essentially "glue code" that converts variables between Java and C and calls the native libraries. The native libraries, in turn, interact with the drivers that control the hardware. In the case of LWJGL, this interaction encompasses everything from rendering visuals and processing audio to supporting various parallel computing tasks.

The use of JNI has both benefits and drawbacks. On the positive side, JNI allows Java applications to access a wide range of native libraries and APIs, enabling them to leverage exotic hardware capabilities that are not directly supported by the JVM. However, JNI also introduces overhead, as calls to native methods are generally slower than calls to Java methods. Additionally, JNI requires developers to write and maintain glue code that bridges the gap between Java and native code, which can be complex and error prone.

Challenges and Limitations

Although JNI has been successful in providing Java developers with access to native APIs, it also highlights some of the challenges and limitations of this approach. One key challenge is the complexity of working with native APIs, which often have different design philosophies and conventions than Java. This can lead to inefficiencies, especially when dealing with low-level details such as memory management and error handling.

Additionally, LWJGL's reliance on JNI to interface with native libraries comes with its own set of challenges. These challenges are highlighted by the insights of Gary Frost, an architect at Oracle, who brings a rich history in bridging Java with exotic hardware:

- **Memory management mismatch:** The JVM is highly autonomous in managing memory. It controls the memory allocation and deallocation life cycle, providing automatic garbage collection to manage object lifetimes. This control, however, can clash with the behavior of native libraries and hardware APIs that require manual control over memory. APIs for GPUs and other accelerators often rely on asynchronous operations, which can lead to a mismatch with the JVM's memory management style. These APIs commonly allow data movement requests and kernel dispatches to return immediately, with the actual operations being queued for later execution. This latency-hiding mechanism is crucial for achieving peak performance on GPUs and similar hardware, but its asynchronous nature conflicts with the JVM's garbage collector, which may relocate Java objects at any time. The pointers passed to the GPU through JNI may become invalid in mid-operation, necessitating the use of functions like `GetPrimitiveArrayCritical` to "pin" the objects and prevent them from being moved. However, this function can pin objects only until the corresponding JNI call returns, forcing Java applications to artificially wait until data transfers are complete, thereby hindering the performance benefits of asynchronous operations. It's worth noting that this memory management mismatch is not unique to LWJGL. Other Java frameworks and tools, such as Aparapi, and TornadoVM (as of this writing), also face similar challenges in this regard.
- **Performance overhead:** Transitions between Java and native code via JNI are generally slower than pure Java calls due to the need to convert data types and manage different calling conventions. This overhead can be relatively minor for applications that make infrequent calls to native methods. However, for performance-critical tasks that rely heavily on native APIs, JNI overhead can become a substantial performance bottleneck. The forced synchronization between the JVM and native libraries merely exacerbates this issue.
- **Complexity and maintenance:** Using JNI requires a firm grasp of both Java and C/C++ because developers must write "glue code" to bridge the two languages. This need for bilingual proficiency and the translation between differing paradigms can introduce complexities, making writing, debugging, and maintaining JNI code a challenging task.

LWJGL and JNI remain instrumental in allowing Java applications to access native libraries and leverage exotic hardware capabilities, but understanding their nuances is crucial, especially in performance-critical contexts. Careful design and a thorough understanding of both the JVM and the target native libraries can help developers navigate these challenges and harness the full power of exotic hardware through Java. LWJGL provides a baseline against which we can compare other approaches, such as Aparapi, Project Sumatra, and TornadoVM, which we will discuss in the following sections.

Aparapi: Bridging Java and OpenCL

Aparapi, an acronym for "A PARallel API," is a Java API that allows developers to implement parallel computing tasks. It serves as a bridge between Java and OpenCL. Aparapi provides a way

for Java developers to offload computation to GPUs or other OpenCL-capable devices, thereby leveraging the remarkable coordinated computation power of these devices.

Using Aparapi in Java applications involves defining parallel tasks using annotations and classes provided by the Aparapi API. Once these tasks are defined, Aparapi takes care of translating the Java bytecode into OpenCL, which can then be executed on the GPU or other hardware accelerators.

To truly appreciate Aparapi's capabilities, let's venture into the world of astronomy, where an adaptive optics technique is used in telescopes to enhance the performance of optical systems by adeptly compensating for distortions caused by wavefront aberrations.[14] A pivotal element of an adaptive optics system is the deformable mirror, a sophisticated device that is used to correct the distorted wavefronts.

In my work at the Center for Astronomical Adaptive Optics (CAAO), we used adaptive optics systems to correct for atmospheric distortions in real time. This task required processing large amounts of sensor data to adjust the shape of the telescope's mirror—a task that could greatly benefit from parallel computation. Aparapi could potentially be used to offload these computations to a GPU or other OpenCL-capable device, enabling the CAAO team to process the wavefront data in parallel. This would significantly speed up the correction process and allow the team to adjust the telescope more quickly and accurately to changing atmospheric conditions.

```
import com.amd.aparapi.Kernel;
import com.amd.aparapi.Range;

public class WavefrontCorrection {
    public static void main(String[] args) {
        // Assume wavefrontData is a 2D array representing the distorted wavefront
        final float[][] wavefrontData = getWavefrontData();

        // Create a new Aparapi Kernel, a unit of computation designed for wavefront correction
        Kernel kernel = new Kernel() {
          // The run method defines the computation. It will be executed in parallel on the GPU.
            @Override
            public void run() {
               // Get the global id, a unique identifier for each work item (computation)
                int x = getGlobalId(0);
                int y = getGlobalId(1);

                // Perform the wavefront correction computation.
                // This is a placeholder; the actual computation would depend on the specifics of
                   the adaptive optics system
                wavefrontData[x][y] = wavefrontData[x][y] * 2;
                // In this example, each pixel of the wavefront data is simply doubled.
```

[14] https://en.wikipedia.org/wiki/Adaptive_optics

```
                // In real-world applications, implement your wavefront correction algorithm here.
            }
        };

        // Execute the kernel with a Range representing the size of the wavefront data
        // Range.create2D creates a two-dimensional range equal to the size of the wavefront data.
        // This determines the work items (computations) that will be parallelized for the GPU.
        kernel.execute(Range.create2D(wavefrontData.length, wavefrontData[0].length));
    }
}
```

In this code, `getWavefrontData()` is a method that retrieves the current wavefront data from the adaptive optics system. The computation inside the `run()` method of the kernel would be replaced with the actual computation required to correct the wavefront distortions.

NOTE When working with Aparapi, it's essential for Java developers to have an understanding of the OpenCL programming and execution models. This is because Aparapi translates the Java code into OpenCL to run on the GPU. The code snippet shown here demonstrates how a wavefront correction algorithm can be implemented using Aparapi, but numerous intricacies must also be considered in such a use case. For instance, 2D Java arrays might not be directly supported in Aparapi due to the need to flatten memory for GPU execution. Additionally, this code won't execute in a standard sequential Java manner; instead, a wrapper method is required to invoke this code if the function is unoptimized. This highlights the importance of being aware of the underlying hardware and execution models when leveraging Aparapi for GPU acceleration.

Aparapi and the JVM

Aparapi offers a seamless integration with the JVM by converting Java bytecode into executable OpenCL code that can be executed on the GPU or other hardware accelerators. This translation is performed by the Aparapi compiler and runtime, which are optimized for OpenCL. The JVM then offloads the execution of this code to the GPU via Aparapi and OpenCL, enabling Java applications to leverage significant performance improvements for data parallel workloads.

Figure 9.2 illustrates the flow from a Java application using Aparapi to computation offloading. The left side of the diagram shows the JVM, within which our Java application operates. The application (on the right) utilizes Aparapi to define a *Kernel*[15] that performs the computation we want to offload. The "OpenCL-Enabled Compiler and Runtime" then translates this *Kernel* from Java bytecode into OpenCL code. The resultant OpenCL code is executed on the GPU or another OpenCL-capable device, as depicted by the OpenCL arrow leading to the GPU.

[15] https://www.javadoc.io/doc/com.aparapi/aparapi/1.9.0/com/aparapi/Kernel.html

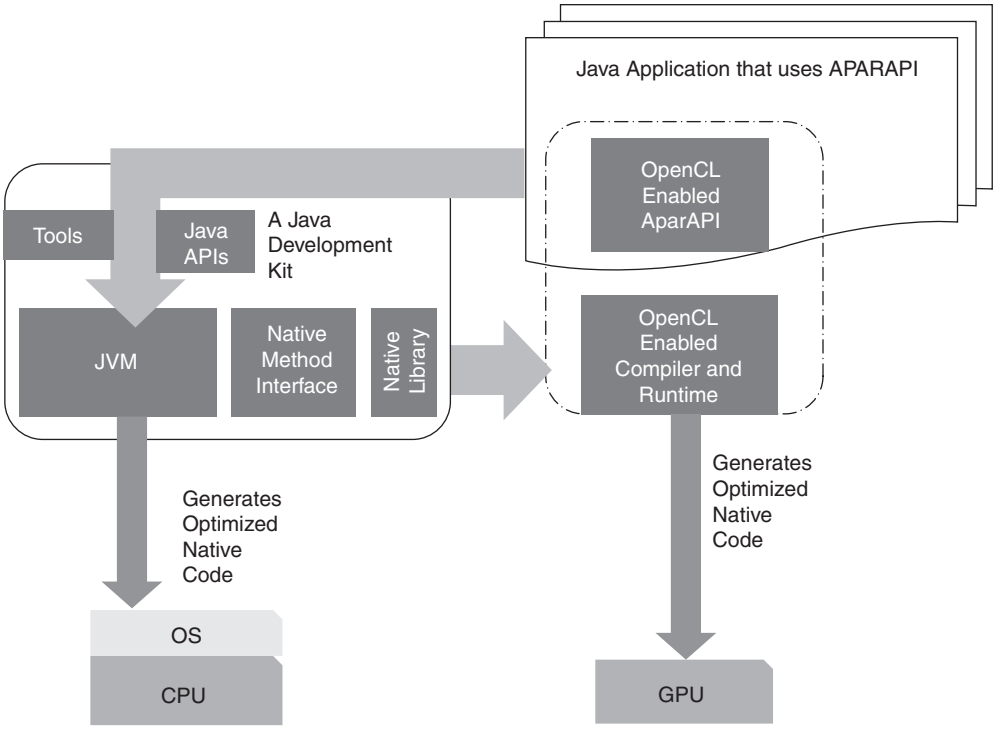

Figure 9.2 Aparapi and the JVM

Challenges and Limitations

Aparapi offers a promising avenue for Java developers to harness the computational prowess of GPUs and other hardware accelerators. However, like any technology, it presents its own set of challenges and limitations:

- **Data transfer bottleneck:** One of the main challenges lies in managing the data transfer between the CPU and the GPU. Data transfer can be a significant bottleneck, especially when working with large amounts of data. Aparapi provides mechanisms such as the ability to specify array access types (read-only, write-only, or read-write on the GPU) to help optimize the data transfer.
- **Explicit memory management:** Java developers are accustomed to automatic memory management courtesy of the garbage collector. In contrast, OpenCL demands explicit memory management, introducing a layer of complexity and potential error sources. Aparapi's approach, which translates Java bytecode into OpenCL, does not support dynamic memory allocation, further complicating this aspect.

- **Memory limitations:** Aparapi cannot exploit the advantages of constant or local memory, which can be crucial for optimizing GPU performance. In contrast, TornadoVM automatically leverages these memory types through JIT compiler optimizations.[16]
- **Java subset and anti-pattern:** Aparapi supports only a subset of the Java language. Features such as exceptions and dynamic method invocation are not supported, requiring developers to potentially rewrite or refactor their code. Furthermore, the Java kernel code, while needing to pass through the Java compiler, is not actually intended to run on the JVM. As Gary Frost puts it, this approach of using Java syntax to represent "intent" but not expecting the JVM to execute the resulting bytecode represents an anti-pattern in Java and can make it difficult to leverage GPU features effectively.

It's important to recognize that these challenges are not unique to Aparapi. Platforms such as TornadoVM, OpenCL, CUDA, and Intel oneAPI[17] also support specific subsets of their respective languages, often based on C/C++. In consequence, developers working with these platforms need to be aware of the specific features and constructs that are supported and adjust their code accordingly.

In conclusion, Aparapi represents a step forward in enabling Java applications to leverage the power of exotic hardware. Ongoing development and improvements are likely to address current challenges, making it even easier for developers to harness the power of GPUs and other hardware accelerators in their Java applications.

Project Sumatra: A Significant Effort

Project Sumatra was a pioneering initiative in the landscape of JVM and high-performance hardware. Its primary goal was to enhance the JVM's ability to work with GPUs and other accelerators. The project aimed to enable Java applications to offload data parallel tasks to the GPU directly from within the JVM itself. This was a significant departure from the traditional model of JVM execution, which primarily targeted CPUs.

Project Sumatra introduced several key concepts and components to achieve its goals. One of the most significant was the Heterogeneous System Architecture (HSA) and the HSA Intermediate Language (HSAIL). HSAIL is a portable intermediate language that is finalized to hardware ISA at runtime. This enabled the dynamic generation of optimized native code for GPUs and other accelerators.

Furthermore, the coherency between CPU and GPU caches provided by HSA obviated the need for moving data across the system bus to the accelerator, streamlining the offloading process for Java applications and enhancing performance.

Another key component of Project Sumatra was the Graal JIT compiler, which we briefly explored in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM." Graal

[16] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. *Automatically Exploiting the Memory Hierarchy of GPUs Through Just-in-Time Compilation*. Paper presented at the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21), April 16, 2021. https://research.manchester.ac.uk/files/190177400/MPAPADIMITRIOU_VEE2021_GPU_MEMORY_JIT_Preprint.pdf.

[17] www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html