```
    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.

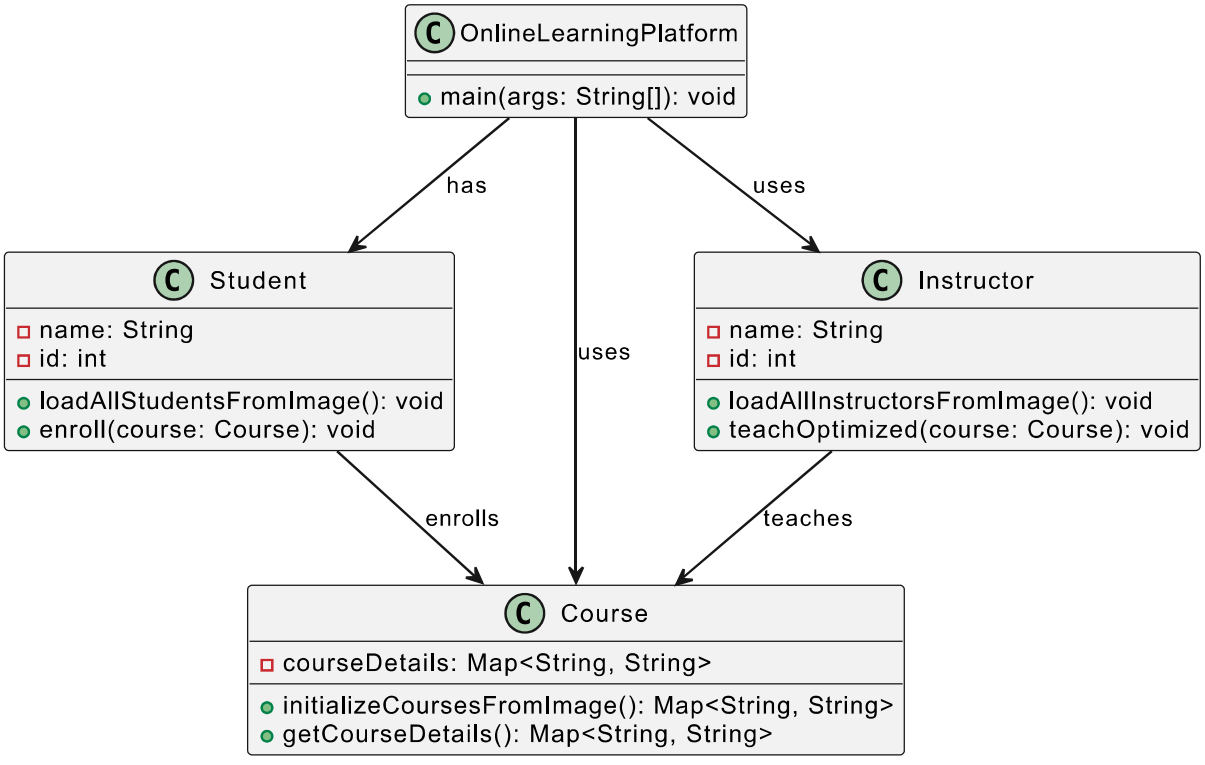

Figure 8.6 New Class Diagram: Post–Project Leyden Optimizations

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.

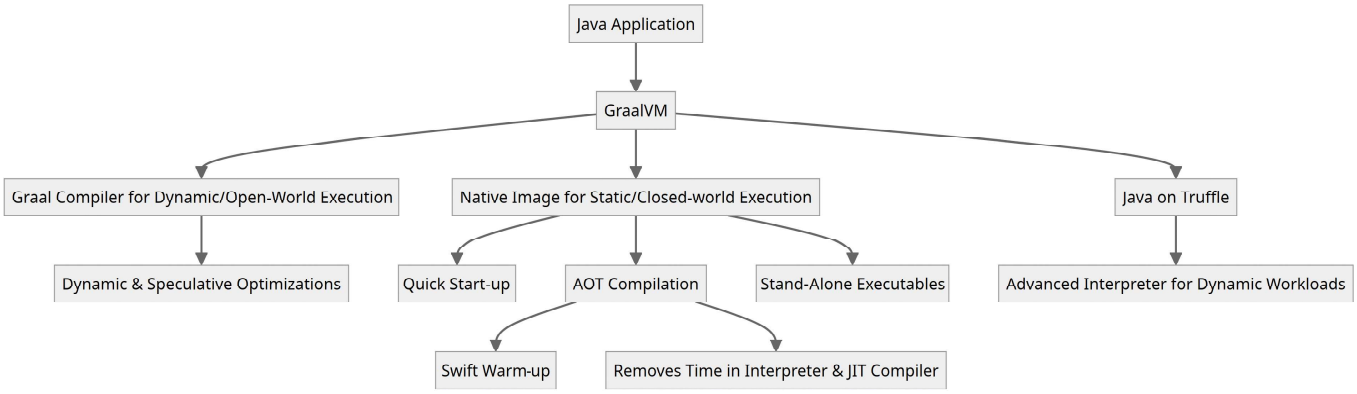

Figure 8.7 GraalVM Optimization Strategies

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.

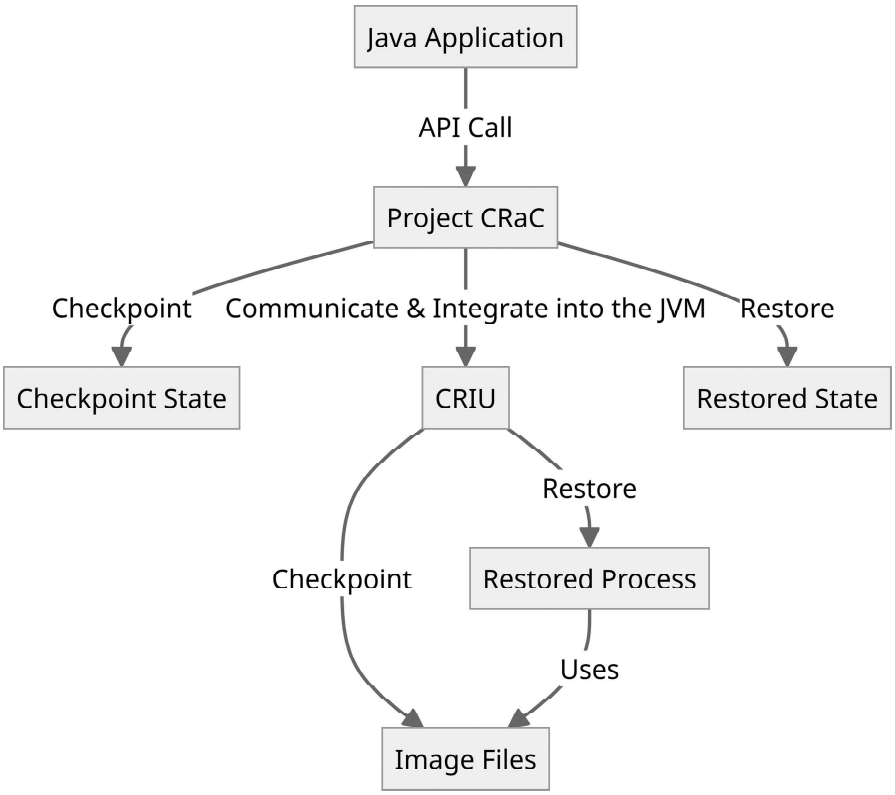

Figure 8.8 Applying Project CRaC and CRIU Strategies to a Java Application

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.

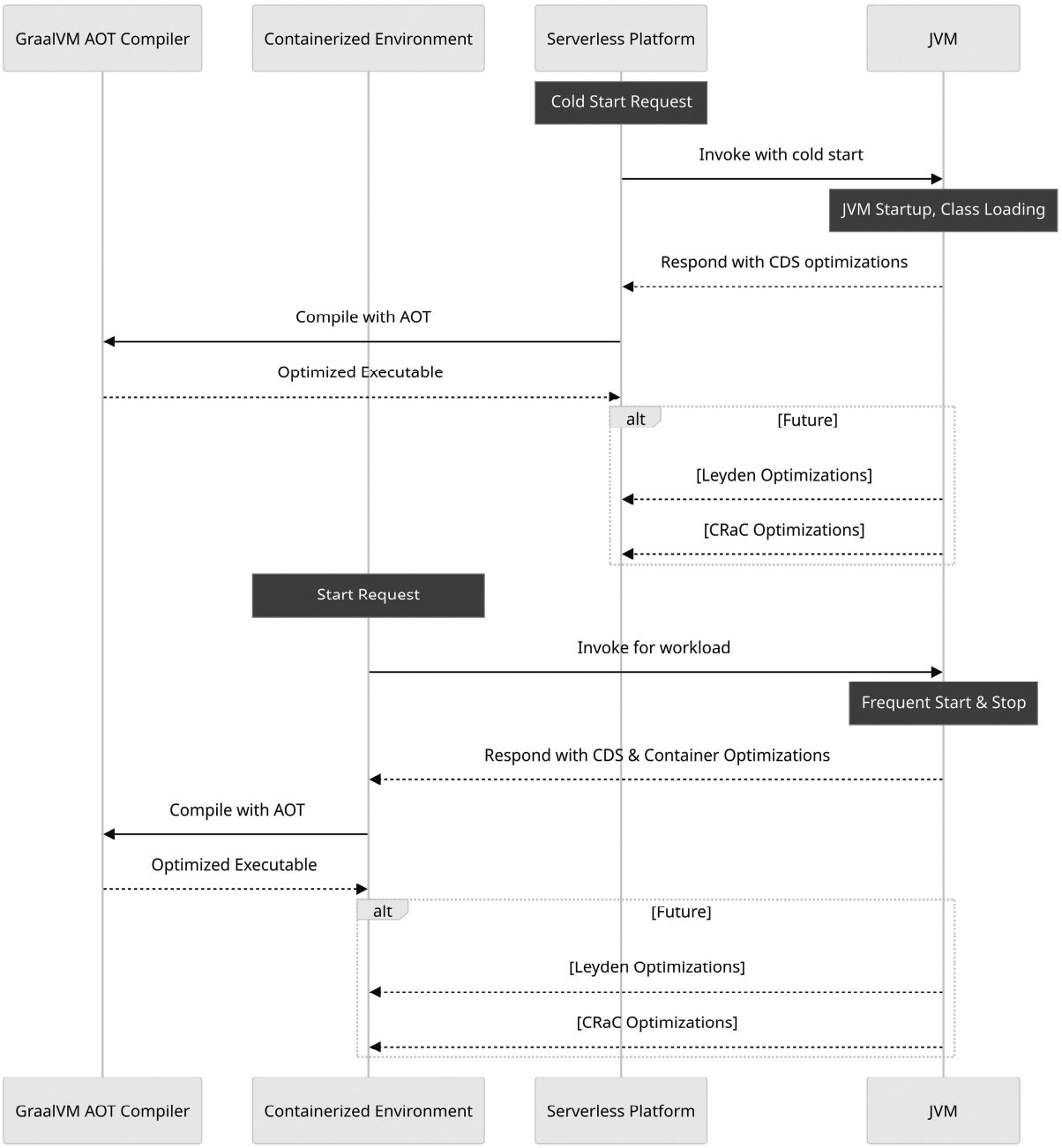

Figure 8.9 JVM Landscape with Containerized and Serverless Platforms

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.

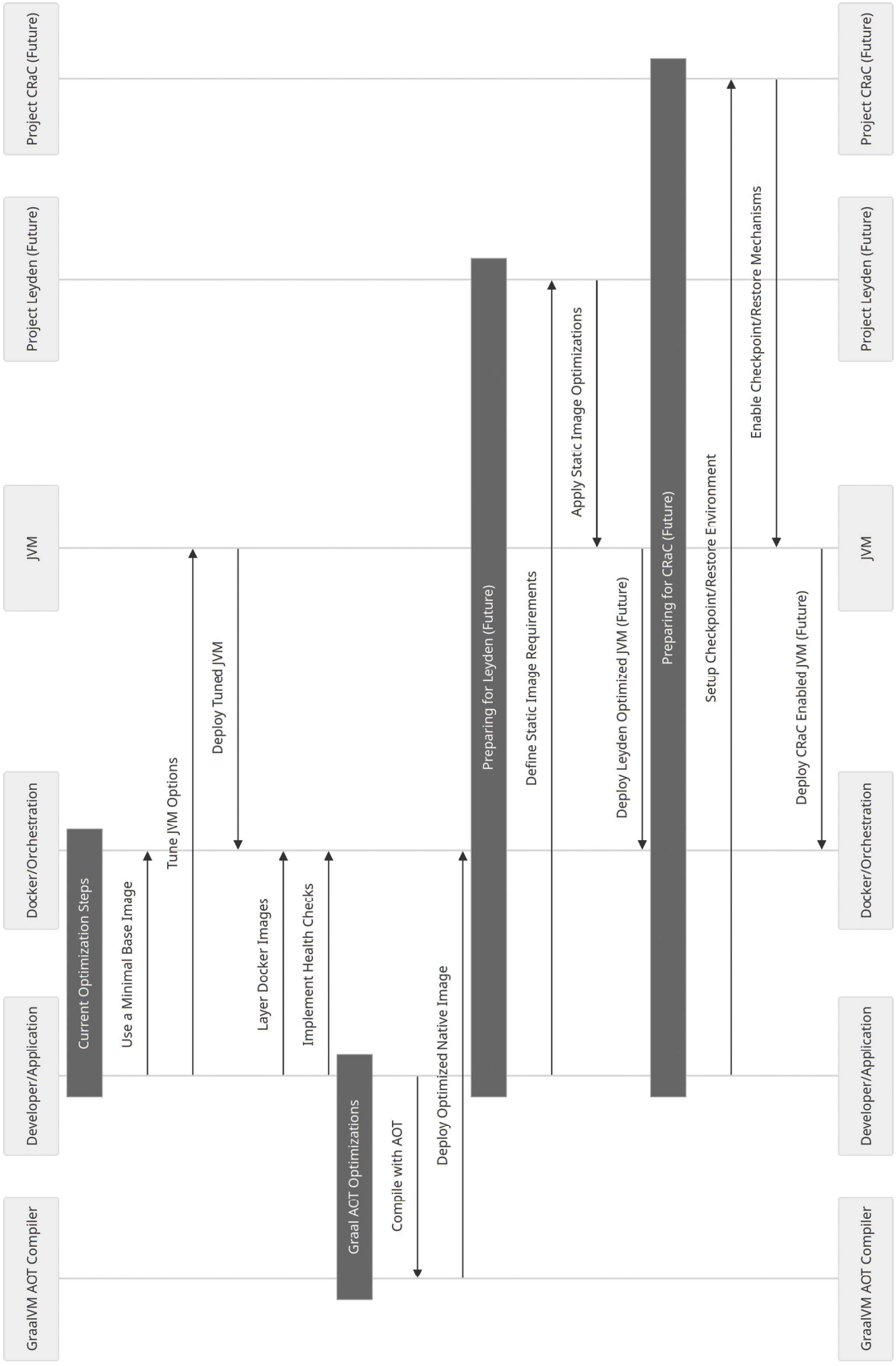

Figure 8.10 Containers' Strategies When Running on the JVM

- **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).

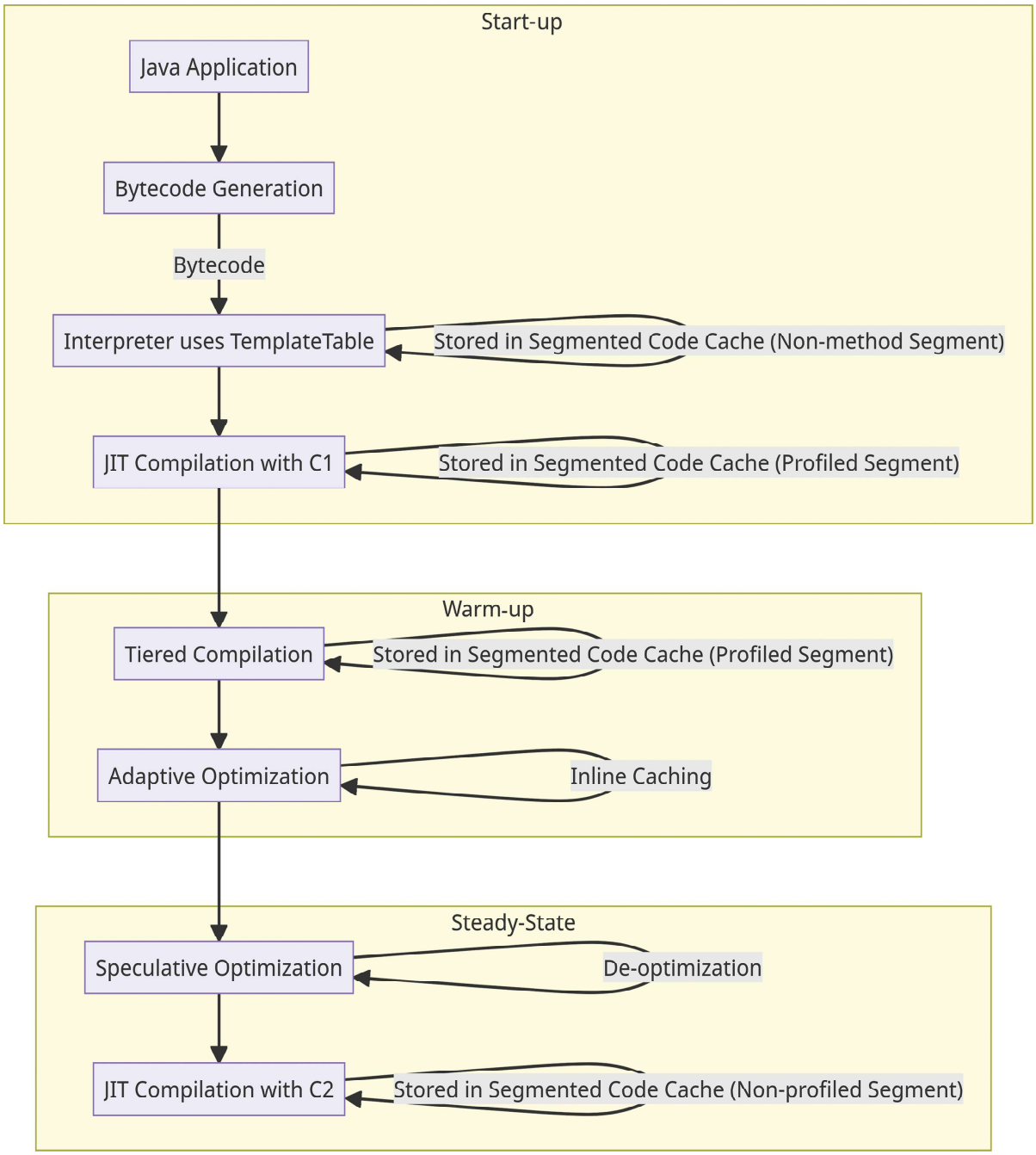

Figure 8.11 OpenJDK Compilation Strategies

- **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.