

is a dynamic compiler that can be used as a JIT compiler within the JVM. Within Sumatra's framework, Graal was used to generate HSAIL code from Java bytecode, which could then be executed on the GPU.

Project Sumatra and the JVM

Project Sumatra was designed to be integrated closely with the JVM. It explored the integration of components such as the “Graal JIT Backend” and “HSA Runtime” into the JVM architecture, as depicted in Figure 9.3. These enhancements to the architecture allowed Java bytecode to be compiled into HSAIL code, which the HSA runtime could use to generate optimized native code that could run on the GPU (in addition to the existing JVM components that generate optimized native code for the CPU).

A prime target of Project Sumatra was the enhancement of the Java 8 Stream API to execute on the GPU, where the operations expressed in the Stream API would be offloaded to the GPU for processing. This approach would be particularly beneficial for compute-intensive tasks such as correcting numerous wavefronts in the astronomy scenario described earlier—the parallelized computation capabilities of the GPU could be leveraged to perform these corrections more quickly than would be possible on the CPU. However, to utilize these enhancements, specialized hardware that supports the HSA infrastructure, like AMD’s Accelerated Processing Units (APUs), was required.

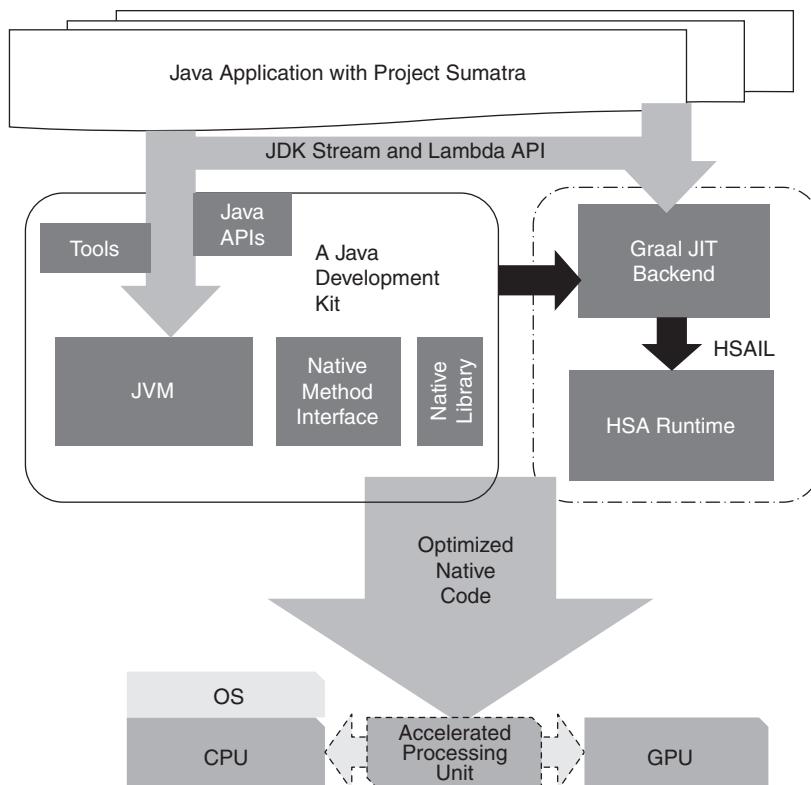


Figure 9.3 Project Sumatra and the JVM

In the following example, we use the Stream API to create a parallel stream from a list of wavefronts. Then we use a lambda expression, `wavefront -> wavefront.correct()`, to correct each wavefront. The corrected wavefronts are collected into a new list.

```
// Assume we have a List of Wavefront objects called wavefronts
List<Wavefront> wavefronts = ...;

// We can use the Stream API to process these wavefronts in parallel
List<Wavefront> correctedWavefronts = wavefronts.parallelStream()
    .map(wavefront -> {
        // Here, we're using a lambda expression to define the correction operation
        Wavefront correctedWavefront = wavefront.correct();
        return correctedWavefront;
    })
    .collect(Collectors.toList());
```

In a traditional JVM, this code would be executed in parallel on the CPU. However, with Project Sumatra, the goal was to allow such data parallel computations to be offloaded to the GPU. The JVM would recognize that this computation could be offloaded to the GPU, generate the appropriate HSAIL code, and execute it on the GPU.

Challenges and Lessons Learned

Despite its ambitious goals, Project Sumatra faced several challenges. One of the main challenges was the complexity of mapping Java's memory model and exception semantics to the GPU. Additionally, the project was closely tied to the HSA, which limited its applicability to HSA-compatible hardware.

A significant part of this project was led by Tom Deneau, Eric Caspole, and Gary Frost. Tom, with whom I had the privilege of working at AMD along with Gary, was the team lead for Project Sumatra. Beyond being just a colleague, Tom was also a mentor, whose deep understanding of the Java memory model and guidance played an instrumental role in Project Sumatra's accomplishments. Collectively, Tom, Eric, and Gary's extraordinary efforts allowed Project Sumatra to push the boundaries of Java's integration with the GPU, implementing advanced features such as thread-local allocation on the GPU and safe pointing from the GPU back to the interpreter.

However, despite the significant progress made, Project Sumatra was ultimately discontinued. The complexity of modifying the JVM to support GPU execution, along with the challenges of keeping pace with rapidly evolving hardware and software ecosystems, led to this decision.

In the realm of JVM and GPU interaction, the J9/CUDA4J offering deserves a noteworthy mention. The team behind J9/CUDA4J decided to use a Stream-based programming model similar to that included in Project Sumatra, and they continue to support their solution to date. Although J9/CUDA4J extends beyond the scope of this book, acknowledging its existence and contribution paints a more comprehensive picture of Java's journey toward embracing GPU computation.

Despite the discontinuation of Project Sumatra, its contributions to the field remain invaluable. It demonstrated the potential benefits of offloading computation to the GPU, while shedding light on the challenges that must be addressed to fully harness these benefits. The knowledge gained from Project Sumatra continues to inform ongoing and future projects in the area of JVM and hardware accelerators. As Gary puts it, we pushed hard on the JVM through Project Sumatra, and you can see its “ripples” in projects like the Vector API, Value Types, and Project Panama.

TornadoVM: A Specialized JVM for Hardware Accelerators

TornadoVM is a plug-in to OpenJDK and GraalVM that allows developers to run Java programs on heterogeneous or specialized hardware. According to Dr. Fumero, TornadoVM’s vision actually extends beyond mere compilation for exotic hardware: It aims to achieve both code and performance portability. This is realized by harnessing the unique features of accelerators, encompassing not just the compilation process but also intricate aspects like data management and thread scheduling. By doing so, TornadoVM seeks to provide a holistic solution for Java developers delving into the realm of heterogeneous hardware computing.

TornadoVM offers a refined API that allows developers to express parallelism in their Java applications. This API is structured around tasks and annotations that facilitate parallel execution. The `TaskGraph`, which is included in the newer versions of TornadoVM, is central to defining the computation, while the `TornadoExecutionPlan` specifies execution parameters. The `@Parallel` annotation can be used to mark methods that should be offloaded to accelerators.

Revisiting our adaptive optics system scenario for a large telescope, suppose we have a large array of wavefront sensor data that requires real-time processing to correct for atmospheric distortions. Here’s an updated example of how you might use TornadoVM in this context:

```
import uk.ac.manchester.tornado.api.TaskGraph;

public class TornadoExample {
    public static void main(String[] args) {
        // Create a task graph
        TaskGraph taskGraph = new TaskGraph("s0")
            .transferToDevice(DataTransferMode.FIRST_EXECUTION, wavefronts)
            .task("t0", TornadoExample::correctWavefront, wavefronts, correctedWavefronts)
            .transferToHost(DataTransferMode.EVERY_EXECUTION, correctedWavefronts);

        // Execute the task graph
        ImmutableTaskGraph itg = taskGraph.snapshot();
        TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(itg);

        executionPlan.execute();
    }
}
```

```

public static void correctWavefront(float[] wavefronts, float[] correctedWavefronts, int N) {
    for (@Parallel int i = 0; i < wavefronts.length; i++) {
        for (@Parallel int j = 0; j < wavefronts[i].length; j++) {
            correctedWavefronts[i * N + j] = wavefronts[i * N + j] * 2;
        }
    }
}

```

In this example, a `TaskGraph` named `s0` is created; it outlines the sequence of operations and data transfers. The `wavefronts` data is transferred to the device (like a GPU) before computation starts. The `correctWavefront` method, added as a task to the graph, processes each `wavefront` to correct it. Post computation, the corrected data is transferred back to the host (like a CPU).

According to Dr. Fumero, TornadoVM (like Aparapi) doesn't support user-defined objects, but rather uses a collection of predefined ones. In our example, `wavefronts` would likely be a `float` array. Java developers using TornadoVM should be acquainted with OpenCL's programming and execution models. The code isn't executed in typical sequential Java fashion. If the function is deoptimized, a wrapper method is needed to invoke this code. A challenge arises when handling 2D Java arrays: The memory might require flattening unless the GPU can re-create the memory layout—a concern reminiscent of the issues with Aparapi.

TornadoVM and the JVM

TornadoVM is designed to be integrated closely with the JVM. It extends the JVM to exploit the expansive concurrent computation power of specialized hardware, thereby boosting the performance of JVM-based applications.

Figure 9.4 offers a comprehensive summary of the architecture of TornadoVM at a very high level. As you can see this diagram, TornadoVM introduces several new components to the JVM, each designed to optimize Java applications for execution on heterogenous hardware:

- **API (task graphs, execution plans, and annotations):** Developers use this interface to define tasks that can be offloaded to the GPU. Tasks are defined using Java methods, which are annotated to indicate that they can be offloaded, as shown in the earlier example.
- **Runtime (data optimizer + bytecode generation):** The TornadoVM runtime is responsible for optimizing data movement between the host (CPU) and the device (GPU). It uses a task-based programming model in which each task is associated with a data descriptor. The data descriptor provides information about the data size, shape, and type, which TornadoVM uses to manage data transfers between the host and the device. TornadoVM also provides a caching mechanism to avoid unnecessary data transfers. If the data on the device is up-to-date, TornadoVM can skip the data transfer and directly use the cached data on the device.

- **Execution engine (bytecode interpreter + OpenCL +PTX + SPIR-V/LevelZero drivers):** Instead of generating bytecode for execution on the accelerator, TornadoVM uses the bytecode to orchestrate the entire application from the host side. This approach, while an implementation detail, offers extensive possibilities for runtime optimizations, such as dynamic task migration and batch processing, while maintaining a clean design.¹⁸
- **JIT compiler + memory management:** TornadoVM extends the Graal JIT compiler to generate code optimized for GPUs and other accelerators. Recent changes in memory management mean each Java object now possesses its individual memory buffer on the target accelerator, mirroring Aparapi's approach. This shift enhances the system's flexibility, allowing multiple applications to share a single GPU, which is ideal for cloud setups.

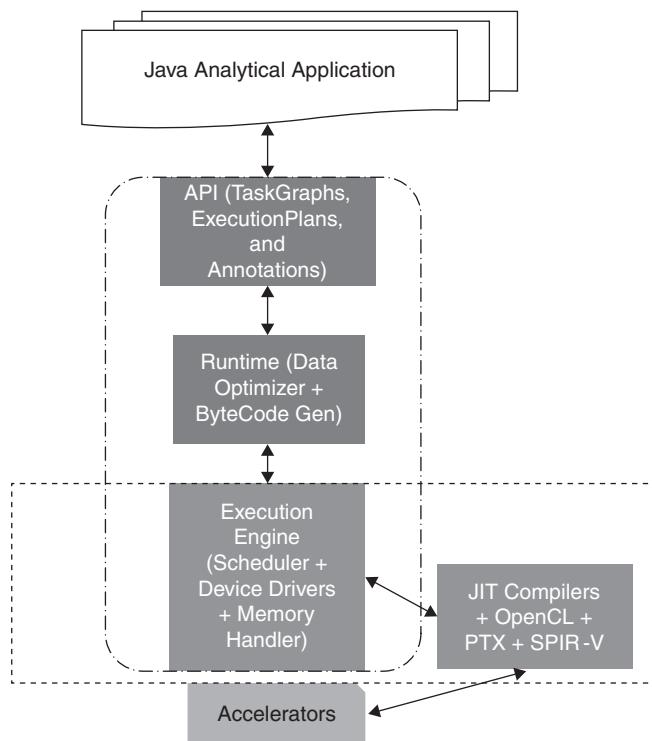


Figure 9.4 TornadoVM

¹⁸ Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. “Dynamic Application Reconfiguration on Heterogeneous Hardware.” In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’19)*, April 14, 2019. https://jjfumero.github.io/files/VEE2019_Fumero_Preprint.pdf.

In a traditional JVM, Java bytecode is executed on the CPU. However, with TornadoVM, Java methods that are annotated as tasks can be offloaded to the GPU or other accelerators. This allows TornadoVM to boost the performance of JVM-based applications.

Challenges and Future Directions

TornadoVM, like any pioneering technology, has faced its share of hurdles. One of the key challenges is the complexity of mapping Java's memory model and exception semantic to the GPU. Dr. Fumero emphasizes that TornadoVM shares certain challenges with Aparapi, such as managing nonblocking operations on GPUs due to the garbage collector. As with Aparapi, data movement between the CPU and accelerators can be challenging. TornadoVM addresses this challenge with a data optimizer that minimizes data transfers. This is possible through the Task-Graph API, which can accommodate multiple tasks, with each task pointing to an existing Java method.

TornadoVM continues to evolve and adapt, pushing the boundaries of what's possible with the JVM and exotic hardware. It serves as a testament to the potential benefits of offloading computation to the GPU and underscores the challenges that need to be surmounted to realize these benefits.

Project Panama: A New Horizon

Project Panama has made significant strides in the domain of Java performance optimization for specialized hardware. It is designed to enhance the JVM's interaction with foreign functions and data, particularly those associated with hardware accelerators. This initiative marks a notable shift from the conventional JVM execution model, historically centered on standard processors.

Figure 9.5 is a simplified block diagram showing the key components of Project Panama—the Vector API and the FFM API.

NOTE For the purpose of this chapter, and this section in particular, I have used the latest build available as of writing this book: JDK 21 EA.

The Vector API

A cornerstone of Project Panama is the Vector API, which introduces a new way for developers to express computations that should be carried out over vectors (that is, sequences of values of the same type). Specifically, this interface is designed to perform vector computations that are compiled at runtime to optimal vector hardware instructions on compatible CPUs.

The Vector API facilitates the use of Single Instruction Multiple Data (SIMD) instructions, a type of parallel computing instruction set. SIMD instructions allow a single operation to be performed on multiple data points simultaneously. This is particularly useful in tasks such as graphics processing, where the same operation often needs to be performed on a large set of data.

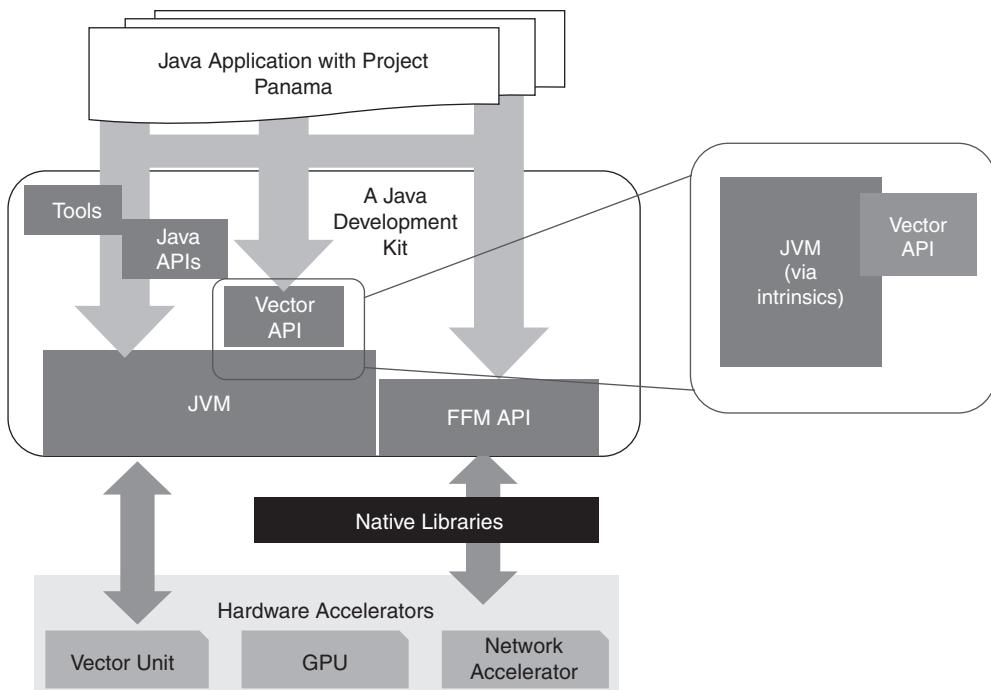


Figure 9.5 Project Panama

As of this book's writing, the Vector API resides in the `jdk.incubator.vector` module in the Project Panama early-access builds. It provides several benefits:

- **Vector computation expression:** The API offers a suite of vector operations that can be employed to articulate computations over vectors. Each operation is applied element-wise, ensuring uniformity in the processing of each element in the vector.
- **Optimal hardware instruction compilation → enhanced portability:** The API is ingeniously designed to compile these computations at runtime into the most efficient vector hardware instructions on applicable architectures. This unique feature allows the same Java code to leverage vector instructions across a diverse range of CPUs, without necessitating any modifications.
- **Leveraging SIMD instructions → performance boost:** The API harnesses the power of SIMD instructions, which can dramatically enhance the performance of computations over large data sets.
- **Correspondence of high-level operations to low-level instructions → increased developer productivity:** The API is architected in such a way that high-level vector operations correspond directly to low-level SIMD instructions via intrinsics. Thus, developers can write code at a high level of abstraction, while still reaping the performance benefits of low-level hardware instructions.

- **Scalability:** The Vector API offers a scalable solution for processing vast amounts of data in a concurrent manner. As data sets continue to expand, the ability to perform computations over vectors becomes increasingly crucial.

The Vector API proves particularly potent in the domain of graphics processing, where it can swiftly apply operations over large pixel data arrays. Such vector operations enable tasks like image filter applications to be executed in parallel, significantly accelerating processing times. This efficiency is crucial when uniform transformations, such as color adjustments or blur effects, are required across all pixels.

Before diving into the code, let's clarify the `factor` used in our example. In image processing, applying a filter typically involves modifying pixel values—`factor` represents this modification rate. For instance, a `factor` less than 1 dims the image for a darker effect, while a value greater than 1 brightens it, enhancing the visual drama in our *Shrek: The Musical* game.

With this in mind, the following example demonstrates the application of a filter using the Vector API in our game development:

```
import jdk.incubator.vector.*;

public class ImageFilter {
    public static void applyFilter(float[] shrekPixels, float factor) {
        // Get the preferred vector species for float
        var species = FloatVector.SPECIES_PREFERRED;
        // Process the pixel data in chunks that match the vector species length
        for (int i = 0; i < shrekPixels.length; i += species.length()) {
            // Load the pixel data into a vector
            var musicalVector = FloatVector.fromArray(species, shrekPixels, i);
            // Apply the filter by multiplying the pixel data with the factor
            var result = musicalVector.mul(factor);
            // Store the result back into the pixel data array
            result.toIntArray(shrekPixels, i);
        }
    }
}
```

Through the code snippet shown here, `FloatVector.SPECIES_PREFERRED` allows our filter application to scale across different CPU architectures by leveraging the widest available vector registers, optimizing our SIMD execution. The `mul` operation then methodically applies our intended filter effect to each pixel, adjusting the image's overall brightness.

As the Vector API is evolving, it currently remains in the *incubator* stage—the `jdk.incubator.vector` package is part of the `jdk.incubator.vector` module in JDK 21. Incubator modules are modules that hold features that are not yet standardized but are made available for developers to try out and provide feedback. It is through this iterative process that robust features are refined and eventually standardized.

To get hands-on with the `ImageFilter` program and explore these capabilities, some changes to the build configuration are necessary. The Maven `pom.xml` file needs to be updated to ensure

that the `jdk.incubator.vector` module is included during compilation. The required configuration is as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <source>21</source>
        <target>21</target>
        <compilerArgs>
          <arg>--add-modules</arg>
          <arg>jdk.incubator.vector</arg>
        </compilerArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The Foreign Function and Memory API

Another crucial aspect of Project Panama is the FFM API (also called the FF&M API), which allows a program written in one language to call routines or use services written in another program. Within the scope of Project Panama, the FFM API allows Java code to interoperate seamlessly with native libraries. This glue-less interface to foreign code also supports direct calls to foreign functions and is integrated with the JVM's method handling and linking mechanisms. Thus, it is designed to supersede the JNI by offering a more robust, Java-centric development model, and by facilitating seamless interactions between Java programs and code or data that resides outside the Java runtime.

The FFM API offers a comprehensive suite of classes and interfaces that empower developers to interact with foreign code and data. As of the writing of this book, it is part of the `java.lang.foreign` module in the JDK 21 early-access builds. This interface provides tools that enable client code in libraries and applications to perform the following functions:

- **Foreign memory allocation:** FFM allows Java applications to allocate memory space outside of the Java heap. This space can be used to store data that will be processed by foreign functions.
- **Structured foreign memory access:** FFM provides methods for reading from and writing to the allocated foreign memory. This includes support for structured memory access, so that Java applications can interact with complex data structures in foreign memory.

- **Life-cycle management of foreign resources:** FFM includes mechanisms for tracking and managing the life cycle of foreign resources. This ensures that memory is properly deallocated when it is no longer needed, preventing memory leaks.
- **Foreign function invocation:** FFM allows Java applications to directly call functions in foreign libraries. This provides a seamless interface between Java and other programming languages, enhancing interoperability.

The FFM API also brings about several nonfunctional benefits that can enhance the overall development experience:

- **Better performance:** By enabling direct invocation of foreign functions and access to foreign memory, FFM bypasses the overhead associated with JNI. This results in performance improvements, especially for applications that rely heavily on interactions with native libraries.
- **Security and safety measures:** FFM provides a safer avenue for interacting with foreign code and data. Unlike JNI, which can lead to unsafe operations if misused, FFM is designed to ensure safe access to foreign memory. This reduces the risk of memory-related errors and security vulnerabilities.
- **Better reliability:** FFM provides a more robust interface for interacting with foreign code, thereby increasing the reliability of Java applications. It mitigates the risk of application crashes and other runtime errors that can occur when using JNI.
- **Ease of development:** FFM simplifies the process of interacting with foreign code. It offers a pure-Java development model, which is more intuitive to use and understand than JNI. This facilitates developers in writing, debugging, and maintaining code that interacts with native libraries.

To illustrate the use of the FFM API, let's consider another example from the field of adaptive optics. Suppose we're working on a system for controlling the secondary mirror of a telescope, and we want to call a native function to adjust the mirror. Here's how we could do this using FFM:

```

import java.lang.foreign.*;
import java.util.logging.*;

public class MirrorController {
    public static void adjustMirror(float[] secondaryMirrorAdjustments) {
        // Get a lookup object from the native linker
        var lookup = Linker.nativeLinker().defaultLookup();
        // Find the native function "adjustMirror"
        var adjustMirrorSymbol = lookup.find("adjustMirror").get();
        // Create a memory segment from the array of adjustments
        var adjustmentArray = MemorySegment.ofArray(secondaryMirrorAdjustments);
        // Define the function descriptor for the native function
        var function = FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);
    }
}

```

```

    // Get a handle to the native function
    var adjustMirror=Linker.nativeLinker().downcallHandle(adjustMirrorSymbol,function);
    try {
        // Invoke the native function with the address of the adjustment array
        adjustMirror.invokeExact(adjustmentArray.address());
    } catch (Throwable ex) {
        // Log any exceptions that occur
        Logger.getLogger(MirrorController.class.getName()).log(Level.SEVERE, null, ex);
    }
}
}

```

In this example, we're adjusting the secondary mirror using the `secondaryMirrorAdjustments` array. The code is written using the FFM API in JDK 21.¹⁹ To use this feature, you need to enable preview features in your build system. For Maven, you can do this by adding the following code to your `pom.xml`:

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <compilerArgs>--enable-preview</compilerArgs>
                <release>21</release>
            </configuration>
        </plugin>
    </plugins>
</build>

```

The evolution of the FFM API from JDK 19 to JDK 21 demonstrates a clear trend toward simplification and enhanced usability. A notable change is the shift from using the `SegmentAllocator` class for memory allocation to the more streamlined `MemorySegment.ofArray()` method. This method directly converts a Java array into a memory segment, significantly reducing the complexity of the code, enhancing its readability, and making it easier to understand. As this API continues to evolve, further changes and improvements are likely in future JDK releases.

Challenges and Future Directions

Project Panama is still evolving, and several challenges and areas of future work remain to be addressed. One of the main challenges is the task of keeping pace with the ever-evolving hardware and software technologies. Staying abreast of the many changes in this dynamic field

¹⁹The FFM API is currently in its third preview mode; that means this feature is fully specified but not yet permanent, so it's subject to change in future JDK releases.

requires continual updates to the Vector API and FFM API to ensure they remain compatible with these devices.

For example, the Vector API needs to be updated as new vector instructions are added to CPUs, and as new types of vectorizable data are introduced. Similarly, the FFM API needs to be updated as new types of foreign functions and memory layouts are introduced, and as the ways in which these functions and layouts are used and evolved.

Another challenge is the memory model. The Java Memory Model is designed for on-heap memory, but Project Panama introduces the concept of off-heap memory. This raises questions about how to ensure memory safety and how to integrate off-heap memory with the garbage collector.

In terms of future work, one of the main areas of focus is improving the performance of the Vector and FFM APIs. This includes optimizing the JIT compiler to generate more efficient code for vector operations and improving the performance of foreign function calls and memory accesses.

Another area under investigation is improving the usability of the APIs. This includes providing better tools for working with vector data and foreign functions and improving the error messages and debugging support.

Finally, ongoing work seeks to integrate Project Panama with other parts of the Java ecosystem. This includes integrating it with the Java language and libraries, and with tools such as the Java debugger and profiler.

Envisioning the Future of JVM and Project Panama

As we stand on the precipice of technological advancement, the horizon promises that a transformative era for the JVM and Project Panama lies ahead. Drawing from my experience and understanding of the field, I'd like to share my vision for the future. Figure 9.6 illustrates one possible use case, in which a gaming application utilizes accelerators via the FFM and Vector APIs with the help of high-level JVM-language APIs.

High-Level JVM-Language APIs and Native Libraries

I anticipate the development of advanced JVM-language APIs designed to interface directly with native libraries. A prime example is the potential integration with NVIDIA's RAPIDS project, which is a suite of software libraries dedicated to data science and analytics pipelines on GPUs.²⁰ RAPIDS leverages NVIDIA's CUDA technology, a parallel computing platform and API model, to optimize low-level compute operations on CUDA-enabled GPUs. By developing JVM APIs that can interface with such native APIs, we could potentially simplify the development process, ensuring efficient hardware utilization. This would allow a broader range of developers—including those who may not have deep expertise in low-level hardware programming—to harness the power of hardware accelerators.

²⁰ "RAPIDS Suite of AI Libraries." NVIDIA Developer. <https://developer.nvidia.com/rapids>.

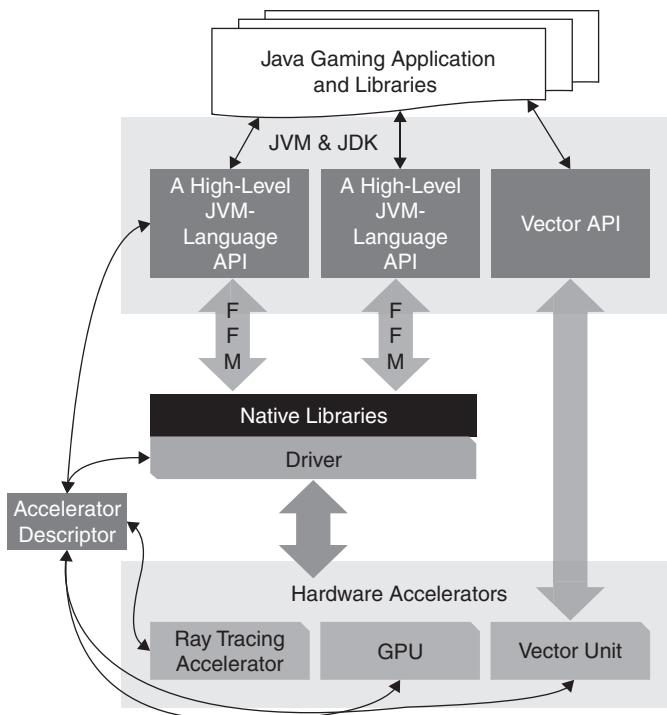


Figure 9.6 A Gaming Application Utilizing Accelerators via the FFM API and Vector API with the Help of High-Level JVM-Language APIs

Vector API and Vectorized Data Processing Systems

The Vector API, when synergized with vectorized data processing systems such as Apache Spark and vector databases, has the potential to revolutionize analytical processing. By performing operations on whole data vectors concurrently rather than on discrete data points, it can achieve significant speed improvements. This API's hardware-agnostic optimizations promise to further enhance such processing, allowing developers to craft efficient code suitable for a diverse array of hardware architectures:

- **Vector databases and Vector API:** Vector databases offer scalable search and analysis of high-dimensional data. The Vector API, with its platform-agnostic optimizations, could further scale these operations.
- **Analytical queries, Apache Spark, and Vector API:** Spark utilizes vectorized operations within its query optimizer for enhanced performance. Integrating the Vector API into this process could further accelerate the execution of analytical queries, capitalizing on the API's ability to optimize across different hardware architectures.

- **Parquet columnar storage file format:** The Parquet columnar storage file format for the Hadoop ecosystem could potentially benefit from the Vector API. The compressed format of Parquet files could be efficiently processed using vector operations, potentially improving the performance of data processing tasks across the Hadoop platform.

Accelerator Descriptors for Data Access, Caching, and Formatting

Within this envisioned future, accelerator descriptors stand out as a key innovation. These metadata frameworks aim to standardize data access, caching, and formatting specifications for hardware accelerator processing. Functioning as a blueprint, they would guide the JVM in fine-tuning data operations to the unique traits and strengths of each accelerator type. The creation of such a system is not just a technical challenge—it requires anticipatory thinking about the trajectory of data-centric computing. By abstracting these specifications, developers could more readily tailor their applications to exploit the full spectrum of hardware accelerators available, simplifying what is currently a complex optimization process.

To realize this vision, meticulous design and collaborative implementation are essential. The Java community must unite in both embracing and advancing these enhancements. In doing so, we can ensure that the JVM not only matches the pace of technological innovation but also redefines benchmarks for performance, usability, and flexibility.

This endeavor is about harmonizing the JVM's solid foundation with the capabilities of cutting-edge hardware accelerators, aiming for significant performance gains without complicating the platform's inherent versatility. Maintaining the JVM's general-purpose nature while optimizing for modern hardware is a delicate balance and a prime focus for Project Panama and the JVM community at large. With the appropriate developments, we are poised to enter a new epoch where the JVM excels in high-performance scenarios, fully leveraging the potential of modern hardware accelerators.

The Future Is Already Knocking at the Door!

As I ventured deeper into the intricate world of JVM and hardware acceleration, it was both enlightening and exhilarating to discover a strong synergy between my vision and the groundbreaking work of Gary Frost and Dr. Fumero. Despite our diverse research trajectories, the collective revelations at JVMLS 2023 showcased a unified vision of the evolution of this domain.

Gary's work with Hardware Accelerator Toolkit (HAT)²¹ was a testament to this forward-thinking vision. Built on the foundation of the FFM API, HAT is more than just a toolkit—it's an adaptable solution across various environments. It includes the *ndrange* API,²² FFM data-wrapping patterns, and an abstraction for vendor-specific runtimes to facilitate the use of hardware accelerators.

The *ndrange* API in HAT, drawing inspiration from TornadoVM, further enhances these capabilities. Complementing HAT's offerings, Project Panama stands out for its powerful functional API and efficient data management strategies. Considering that GPUs require specific data layouts, Project Panama excels in creating GPU-friendly data structures.

²¹www.youtube.com/watch?v=lbKBu3lTftc

²²<https://man.opencl.org/ndrange.html>

Project Babylon²³ is an innovative venture emerging as a pivotal development for enhancing the integration of Java with diverse programming models, including GPUs and SQL. It employs code reflection to standardize and transform Java code, enabling effective execution on diverse hardware platforms. This initiative, which complements Project Panama's efforts in native code interoperation, signifies a transformative phase for Java in leveraging advanced hardware capabilities.

Another highlight of the discussion at JVMS 2023 was Dr. Fumero's presentation on TornadoVM's innovative vision for a Hybrid API.²⁴ As he explained, this API seamlessly merges the strengths of native and JIT-compiled code, enabling developers to tap into speed-centric, vendor-optimized libraries. The seamless integration of Project Panama with this Hybrid API ensures uninterrupted data flow and harmonizes JIT-compiled tasks with library calls, paving the way for a more cohesive and efficient computational journey.

In conclusion, our individual journeys in exploring JVM, Project Panama, and hardware acceleration converge on a shared destination, emphasizing the immense potential of this frontier. We stand on the brink of a transformative era in computing, one in which the JVM will truly unlock the untapped potential of modern hardware accelerators.

Concluding Thoughts: The Future of JVM Performance Engineering

As we draw this comprehensive exploration of JVM performance engineering to a close, it's essential to reflect on the journey we've undertaken. From the historical evolution of Java and its virtual machine to the nuances of its type system, modularity, and memory management, we've traversed the vast landscape of JVM intricacies. Each chapter has been a deep dive into specific facets of the JVM, offering insights, techniques, and tools to harness its full potential.

The culmination of this journey is the vision of the future—a future where the JVM doesn't just adapt but thrives, leveraging the full might of modern hardware accelerators. Our discussions of Project Panama, the Vector API, and more have painted a vivid picture of what lies ahead. It's a testament to the power of collaboration and shared vision.

The tools, APIs, and frameworks we've discussed are the building blocks of tomorrow, and I encourage you to explore them while applying your own expertise, insights, and passion. The JVM community is vibrant, and ever evolving. Your contributions, experiments, and insights we collectively share will shape its future trajectory. Engage with this community. Dive deep into the new tools, push their boundaries, and share your findings. Every contribution, every line of code, every shared experience adds a brick to the edifice we're building.

*Thank you for joining me on this enlightening journey.
Let's continue to explore, innovate, and shape the future of
JVM performance engineering together.*

²³<https://openjdk.org/projects/babylon/>

²⁴Juan Fumero. *From CPU to GPU and FPGAs: Supercharging Java Applications with TornadoVM*. Presentation at JVM Language Summit 2023, August 7, 2023. <https://github.com/jjfumero/jjfumero.github.io/blob/master/files/presentations/TornadoVM-JVMS23v2-clean.pdf>.

Index

Numerics

4-9s, response times, 124–126

5-9s, response times, 125–126

Symbols

@Benchmark, 171

@BenchmarkMode, 171

@Fork, 171

@Measurement, 168–169, 171

@OperationsPerInvocation, 169, 171

@OutputTimeUnit, 171

@Setup, 171

@State, 171

@Teardown, 171

@Warmup, 168–169, 171

_ (underscore), code readability, 51

A

AArch64 ports

macOS/AArch64 port, 40

windows/AArch64 port, 39

accelerators, 307–309

Aparapi, 308, 314, 317–321

cloud computing, 309–312

CUDA4J, 314

descriptors, 335

HAT, 335–336

language design, 313–314

limitations, 311–312

LWJGL, 313–317

- OpenCL, 308
- performance, 311
- Project Panama**, 314
 - FFM API, 309, 330–333
 - vector API, 308, 327–330, 334–335
- Project Sumatra**, 314, 321–324
- resource contention, 311
- toolchains, 313–314
- TornadoVM, 308, 314, 324–327, 336
- adaptive optimization, Hotspot VM**, 9
- adaptive sizing**
 - footprint, performance engineering, 123
 - heap management, 184
- Adaptive spinning (Java 7)**, 243
- Ahead-of-Time (AOT) compilation**
 - GraalVM, 284–285, 298
 - state management, 283–285
- allocation-stall-based GC (Garbage Collection)**, 207
- AMD, infinity fabric**, 144
- analytics**
 - benchmarking performance, 161
 - OLAP, 214–215
- annotations**, 43, 50–51
 - JMH, 169–172
 - meta-annotations, 51
- AOT (Ahead-of-Time) compilation**
 - GraalVM, 284–285, 298
 - state management, 283–285
- Apache**
 - Cassandra, 214–215
 - Flink, 215
 - Hadoop
 - availability, 124
 - GC performance, 214
 - Vector API, 335
 - HBase, 214–215
 - Hive, 214
- Ignite**, 215
- Spark**
 - GC collection performance, 214
 - Vector API, 334
- Aparapi**, 308, 314, 317–321
- application profilers, performance engineering**, 157
- Application Programming Interface (API)**
 - Aparapi, 308, 314, 317–321
 - compatibility, 310–311
 - CUDA4J, 314
 - FFM API, 309, 330–333
 - Hybrid API, 336
 - hypervisors, 310–311
 - language API, JVM, 333–334
 - native libraries, JVM, 333–334
 - ndrange API, 335–336
 - ServiceLoader API, 81–83
 - vector API, 308, 327–330, 334–335
 - virtual threads, 267
- applications**
 - life cycle of, 279
 - Project Leyden, 285
 - ramp-up phase, 276–277
 - steady-state phase, 276–277
 - stopping, 278
- Arm, NUMA advances**, 144
- arrays**
 - byte array internals, 60
 - elements, 61–62
 - length of, 61
 - multidimensional arrays, 49
 - types of, 48–49
- asynchronous logging**
 - backpressure management, 111
 - benefits of, 110
 - best practices, 111–113
 - customizing solutions, 111

implementing, 110–111
Log4j2, 110
Logback, 110
 performance, 112–113
 reliability, 112
 unified logging integration, 111
async-profiler, 157
atomicity, 139–141
availability, performance engineering
 MTBF, 124–126
 MTTR, 124–126

B

backpressure, asynchronous logging, 111
backward compatibility, Java, 83–84
barriers, concurrent computing, 139
benchmarking
 contended locks, 248–251
 harnesses, 164–165
 JMH, 172–174
 @Benchmark, 171
 @BenchmarkMode, 171
 @Fork, 171
 @Measurement, 168–169, 171
 @OperationsPerInvocation, 169, 171
 @OutputTimeUnit, 171
 @Setup, 171
 @State, 171
 @Teardown, 171
 @Warmup, 168–169, 171
 annotations, 169, 171–172
 benchmarking modes, 170
 features of, 165
 loop optimizations, 169
 Maven, 166–170
 measurement phase, 168–169
 perfasm, 174
 profilers, 170–171

profiling benchmarks, 174
 running microbenchmarks,
 166–168
 warm-up phase, 168–169
 writing/building microbenchmarks,
 166–168
JVM memory management, 161–164
 loop optimizations, 169
Maven, 166–170
microbenchmarking, 165–174
 modes, 170
performance engineering, 158
 harnesses, 164–165
 iterative approach, 161
JVM memory management,
 161–164
 metrics, 159
microbenchmarking, 165–174
 process (overview), 159–161
 unified logging, 108
 warm-up phase, 168–169

best practices, asynchronous logging,
111–113

bimorphic call sites, 12
bootstrapping, JVM start-up
 performance, 275
bottom-up methodology, performance
engineering, 146–148

byte array internals, 60

bytecode, 2

loading, 275–276
 start-up performance, optimizing,
 275–276
 verifying, 275–276

C

C#, Project Valhalla comparisons, 67–68
C++ Interpreter, 3
Cache-Coherent NUMA (ccNUMA), 144

caches

- accelerator descriptors, 335
- CodeCache**
 - Hotspot VM, 3
 - Segmented CodeCache, 7–8, 300–302
- hardware, 132
- LLC, 137
- SLC, 137
- warming, 276

call tree analysis, contended locks, performance, 251–254

Cassandra (Apache), 214–215

ccNUMA (Cache-Coherent NUMA), 144

Checkpoint/Restore in Userspace (CRIU), 292–295

Class Data Sharing (CDS), 282

classes

- hidden classes, 38–39
- Hotspot VM deoptimization, 9–10
- loading, 9–10, 276
- primitive classes, 65–66
- Project Valhalla, 65–66
- sealed classes, 38–39, 44, 56–57
- types, 47–48
- value classes, 63–66

client compiler (C1), 7

cloud computing

- exotic hardware, 309–312
- limitations, 311–312
- performance, 311
- resource contention, 311

code footprint, 119

code readability, underscore (_), 51

code refactoring, footprint, performance engineering, 122

CodeCache

- Hotspot VM, 3
- Segmented CodeCache, 7–8, 303

colored pointers, ZGC, 197–198

command-line, Project Valhalla, 67

compact strings, 227–236

compiling

- AOT
 - GraalVM, 284–285, 298
 - state management, 283–285
 - JIT, 276, 283–285
 - modules, 72–76

CompletableFuture frameworks, threads, 264–265

concatenating strings, 224–227

concurrent computing

- algorithms, 14
- atomicity**, 139–141
 - barriers, 139
 - fences, 139
 - GC threads, 16–18
 - happens-before relationships, 139–141
- hardware
 - core utilization, 136
 - memory hierarchies, 136–138
 - processors, 136–138
 - SMT, 136
 - software interplay, 131
- memory
 - access in multiprocessor systems, 141
 - models, atomicity, 139–141
- NUMA, 141, 145
 - architecture of, 143
 - ccNUMA, 144
 - components of, 142–143
 - cross-traffic, 143
 - fabric architectures, 144
 - interleaved memory, 143
 - local traffic, 143
 - nodes in modern systems, 142

- threads
- CompletableFuture frameworks, 264–265
 - ForkJoinPool frameworks, 261–264
 - GC threads, 16–18
 - Java Executor Service, 261
 - thread pools, 261
 - thread-per-request model, 261–266
 - thread-per-task model, 259–260
 - thread-stack processing, 39–40
 - virtual threads, 265–270
- volatiles, 139
- ZGC, 198–200
- condensers, Project Leyden, 286**
- containerized environments**
- performance engineering, 155
 - scalability, 297–298
 - start-up performance, 297–298
- contended locks, 239–240**
- Adaptive spinning (Java 7), 243
 - benchmarking, 248–251
 - call tree analysis, 251–254
 - flamegraph analysis, 249–251
 - Improved Contended Locking, 34
 - lock coarsening, 242
 - lock elision, 242
 - monitor enter/exit operations, 243–245
 - PAUSE instructions, 258–259
 - performance engineering, 245–259
 - ReentrantLock (Java 5), 241
 - spin-loop hints, 258–259
 - spin-wait hints, 257–259
 - StampedLock (Java 8), 241–242
- continuations, virtual threads, 270**
- Coordinated Restore at Checkpoint (CRaC), 292–295**
- core scaling, heap management, 185**
- core utilization, concurrent hardware, 136**
- CRaC (Coordinated Restore at Checkpoint), 292–295**
- CRIU (Checkpoint/Restore in Userspace), 292–295**
- cross-traffic, NUMA, 143**
- CUDA4J API (Application Programming Interface), 314**
- customizing asynchronous logging solutions, 111**
-
- D**
-
- data access, accelerator descriptors, 335**
- data lifespan patterns, 216**
- data structure optimization, footprint, 122**
- DDR (Double Data Rate) memory, 137–138**
- debugging, fast/slow, 99**
- decorators, unified logging, 104–105**
- deduplicating strings, 223–224**
- deflated locks, 238–239**
- degradation, graceful, 17**
- deoptimization, Hotspot VM**
- class loading/unloading, 9–10
 - dynamic deoptimization, 9
 - polymorphic call sites, 11–13
- deprecations, Java 17 (Java SE 17), 41**
- Double Data Rate (DDR) memory, 137–138**
- dynamic dumping, shared archive files, 282–283**
-
- E**
-
- encapsulation, Java 17 (Java SE 17), 40–41**
- enumerations, 49–50**
- Epsilon GC (Garbage Collector), 35**
- exotic hardware, 307–309**
- Aparapi, 308, 314, 317–321
 - cloud computing, 309–312
 - CUDA4J, 314
 - descriptors, 335

HAT, 335–336
language design, 313–314
limitations, 311–312
LWJGL, 313–317
OpenCL, 308
performance, 311
Project Panama, 314
FFM API, 309, 330–333
vector API, 308, 327–330, 334–335
Project Sumatra, 314, 321–324
resource contention, 311
toolchains, 313–314
TornadoVM, 308, 314, 324–327, 336
experimental design, performance engineering, 146

F

fabric architectures, NUMA, 144
failure, MTBF, 124–126
fast-debug, 99
fences, concurrent computing, 139
FFM (Foreign Function and Memory) API, 309, 330–333
flamegraph analysis, contended lock performance, 249–251
Flink, Apache, 215
footprint
 performance engineering
 adaptive sizing policies, 123
 code footprint, 119
 code refactoring, 122
 data structure optimization, 122
 JVM parameter tuning, 122
 managing, 120
 memory footprint, 119
 mitigating issues, 122–123
 NMT, 120–122
 non-heap memory, monitoring with NMT, 120–122

physical resources, 120
strings, reducing footprint, 224–236
Foreign Function and Memory (FFM) API, 309, 330–333
@Fork, 171
ForkJoinPool frameworks, threads, 261–264
formatting, accelerator descriptors, 335

G

G1 GC (Garbage-First Garbage Collector), 16, 178–179
deduplicating strings (dedup), 223–224
heap management
 adaptive sizing, 184
 core scaling, 185
 humongous objects handling, 186
 IHOP, 186
 pause-time predictability, 184–185
 regionalized heaps, 184, 186–188
 marking thresholds, 196–197
NUMA-aware memory allocator, 38
optimizing, 193–196
pause responsiveness, 189–193
performance, 188–197, 217
games, LWJGL, 313–317
Garbage Collector (GC), 2
 allocation-stall-based GC, 207
 analytics (OLAP), 214–215
 concurrent algorithms, 14
 concurrent GC threads, 16–18
 concurrent work, 17
 data lifespan patterns, 216
 Epsilon GC, 35
 future trends, 210–212
 G1 GC, 16, 38, 178–179
 deduplicating strings (dedup), 223–224
 heap management, 184–188
 marking thresholds, 196–197

- optimizing, 193–196
 - pause responsiveness, 189–193
 - performance, 188–197, 217
 - graceful degradation, 17
 - high allocation rate-based GC, 206–207
 - high-usage-based GC, 207–208
 - Hotspot VM, 13
 - hybrid applications (HTAP), 215
 - incremental compacting algorithms, 14
 - LDS, 216–217
 - lots of threads, 18
 - MSC, 16, 22
 - NUMA-aware GC, 181–183
 - old-generation collections, 16
 - operational stores (OLTP), 215
 - parallel GC threads, 16–18
 - pauses, 17
 - performance
 - engineering, 154
 - evaluations, 212–216
 - PLAB, 180–181
 - proactive GC, 208–209
 - reclamation triggers, 16
 - scavenge algorithm, 16
 - Shenandoah GC, 16, 37
 - STW algorithms, 14
 - task queues, 17
 - task stealing, 17
 - thread-local handshakes, 14
 - time-based GC, 204–205
 - TLAB, 179–180
 - ultra-low-pause-time collectors, 14
 - warm-up-based GC, 205–206
 - weak generational hypothesis, 14–16
 - young collections, 14–16
 - ZGC, 16, 35, 37–38, 39–40, 178–179
 - advancements, 209–210
 - allocation-stall-based GC, 207
 - colored pointers, 197–198
 - concurrent computing, 198–200
 - high allocation rate-based GC, 206–207
 - high-usage-based GC, 207–208
 - performance, 217
 - phases, 199–201
 - proactive GC, 208–209
 - thread-local handshakes, 198–199
 - time-based GC, 204–205
 - triggering cycles, 204–209
 - warm-up-based GC, 205–206
 - ZPages, 198, 202–204
- generational hypothesis, weak, 14–16**
- generics, 1, 19–22, 25, 43, 51, 64–66**
- GraalVM, 289**
- AOT compilation, 284–285, 298
 - native image generation, 291
 - OpenJDK support, 291
 - TornadoVM, 308, 314, 324–327, 336
- graceful degradation, 17**
-
- ## H
- Hadoop (Apache)**
- availability, 124
 - GC performance, 214
 - Vector API, 335
- happens-before relationships, 139–141, 237**
- hardware**
- accelerators, TornadoVM, 324–327
 - caches, 132
 - concurrent computing
 - core utilization, 136
 - memory hierarchies, 136–138
 - processors, 136–138
 - SMT, 136
 - software interplay, 131
 - exotic hardware, 307–309

- Aparapi, 308, 314, 317–321
- cloud computing, 309–312
- CUDA4J, 314
- language design, 313–314
- limitations, 311–312
- LWJGL, 313–317
- OpenCL, 308
- performance, 311
- Project Panama, 308–309, 314, 327–335
- Project Sumatra, 314, 321–324
- resource contention, 311
- toolchains, 313–314
- TornadoVM, 308, 314, 324–327, 336
- hardware-aware programming, 132
- HAT, 335–336
- heterogeneity, 310
- performance and, 128–131
- software dynamics, 129–131
- subsystems, performance engineering, 156–157
- harnesses, benchmarking, 164–165**
- HashMap, annotations, 51**
- HAT (Hardware Accelerator Toolkit), 335–336**
- HBase (Apache), 214–215**
- heap management**
 - adaptive sizing, 184
 - core scaling, 185
 - G1 GC, 184–188
 - humongous objects handling, 186
 - IHOP, 186
 - nmethod code heaps, 8
 - off-heap forwarding tables, 201
 - pause-time predictability, 184–185
 - regionalized heaps, 184, 186–188
 - ZGC, off-heap forwarding tables, 201
- heterogeneity, hardware, 310**
- hidden classes, 38–39**
- hierarchies, memory, 136–138**
- high allocation rate-based GC (Garbage Collection), 206–207**
- high-usage-based GC (Garbage Collection), 207–208**
- Hive (Apache), 214**
- hops, 28–30**
- host OS, performance engineering, 156**
- Hotspot VM**
 - adaptive optimization, 9
 - C++ Interpreter, 3
 - client compiler (C1), 7
 - CodeCache, 3
 - contended locks, 242–243
 - deoptimization, 9–10
 - class loading/unloading, 9–10
 - polymorphic call sites, 11–13
 - GC, 13
 - concurrent algorithms, 14
 - concurrent GC threads, 16–18
 - concurrent work, 17
 - future trends, 210–212
 - G1 GC, 16, 178–179, 184–197
 - graceful degradation, 17
 - incremental compacting algorithms, 14
 - lots of threads, 18
 - NUMA-aware GC, 181–183
 - old-generation collections, 16
 - parallel GC threads, 16–18
 - pauses, 17
 - performance evaluations, 212–216
 - PLAB, 180–181
 - reclamation triggers, 16
 - scavenge algorithm, 16
 - Shenandoah GC, 16
 - STW algorithms, 14
 - task queues, 17

- task stealing, 17
- thread-local handshakes, 14
- TLAB, 179–180
- ultra-low-pause-time collectors, 14
- weak generational hypothesis, 14–16
- young collections, 14–16
- ZGC, 16, 178–179, 197–210
- interned strings, 221–223
- interpreter, 5
- JIT compiler, 5
- literal strings, 221–223
- mixed-mode execution, 3
- monitor locks, 238–241
- nmethod, 7–8
- performance-critical methods, optimizing, 3–5
- PLAB, 180–181
- print compilation, 5–6
- Project Valhalla, 67
- server compiler (C2), 7
- start-up performance, 300–302
- steady-state phase, 301
- TemplateTable, 3
- tiered compilation, 6–7
- TLAB, 179–180
- unified logging
 - asynchronous logging integration, 111
 - benchmarking, 108
 - decorators, 104–105
 - identifying missing information, 102
 - infrastructure of, 100
 - JDK 11, 113
 - JDK 17, 113
 - levels, 103–104
 - log tags, 101
 - managing systems, 109
 - need for, 99–100
 - optimizing systems, 109
- outputs, 105–106
- performance, 108
- performance metrics, 101
- specific tags, 102
- tools/techniques, 108
- usage examples, 107–108
- warm-up performance, 298
- compilers, 300–303
- optimizing, 300–301
- HTAP (Hybrid Transactional/Analytical Processing), hybrid applications, 215**
- humongous objects handling, heap management, 186**
- Hybrid API, 336**
- hybrid applications (HTAP), 215**
- hypervisors**
 - constraints, 310–311
 - performance engineering, 156
- I**
- Ignite (Apache), 215**
- IHOP (Initiating Heap Occupancy Percent), heap management, 186**
- image generation (native), GraalVM, 291**
- immutable objects, Project Valhalla, 63**
- implementing modular services, 81–83**
- Improved Contended Locking, 34**
- incremental compacting algorithms, 14**
- indy-fication, string concatenation, 224–227**
- infinity fabric, AMD, 144**
- inflated locks, 239–241**
- Initiating Heap Occupancy Percent (IHOP), heap management, 186**
- inline caches, 12**
- inlining, 7**
- instanceof operator, pattern matching, 38**
- Intel, mesh fabric architectures, 144**
- interface types, 45–46, 51–52**

interleaved memory, NUMA, 143

interned strings, 221–223

interpreter, 5

J

JAR Hell versioning problem, 83–91

Java, 1–2

- application profilers, performance engineering, 157

- array types, 48–49

- asynchronous logging

 - backpressure management, 111

 - benefits of, 110

 - best practices, 111–113

 - customizing solutions, 111

 - implementing, 110–111

 - Log4j2, 110

 - Logback, 110

 - performance, 112–113

 - reliability, 112

 - unified logging integration, 111

- backward compatibility, 83–84

- bytecode, 2

- C# comparisons, 67–68

- class types, 47–48

- enhanced performance, 42

- evolution of, 18–42

- GC, 2

 - concurrent algorithms, 14

 - concurrent GC threads, 16–18

 - concurrent work, 17

 - G1 GC, 16

 - graceful degradation, 17

 - Hotspot VM, 13

 - incremental compacting algorithms, 14

 - lots of threads, 18

 - old-generation collections, 16

parallel GC threads, 16–18

pauses, 17

reclamation triggers, 16

scavenge algorithm, 16

Shenandoah GC, 16

STW algorithms, 14

task queues, 17

task stealing, 17

thread-local handshakes, 14

ultra-low-pause-time collectors, 14

weak generational hypothesis, 14–16

young collections, 14–16

ZGC, 16

Hotspot VM

- adaptive optimization, 9

- C++ Interpreter, 3

- client compiler (C1), 7

- CodeCache, 3

- deoptimization, 9–10, 9–13

- GC, 13–14, 13–18

- interpreter, 5

- JIT compiler, 5

- mixed-mode execution, 3

- nmethod, 7–8

- performance-critical methods, 3–5

- print compilation, 5–6

- server compiler (C2), 7

- TemplateTable, 3

 - tiered compilation, 6–7

interface types, 45–46

JIT compiler, 3, 5

Kotlin comparisons, 68

literals, 45

null, 45

primitive data types, 44–45

reference types, 45–49

release cadence, 34

Scala comparisons, 68

- as statically type-checked language, 43
- as strongly typed language, 43
- Java 1.1 (JDK 1.1), 18–19**
 - array types, 48–49
 - class types, 47–48
 - interface types, 45–46
 - reference types, 45–49
- Java 1.4.2 (J2SE 1.4.2), 18–19**
 - array types, 48–49
 - class types, 47–48
 - interface types, 45–46
 - reference types, 45–49
- Java 5 (J2SE 5.0)**
 - annotations, 43, 50
 - enumerations, 49–50
 - generics, 43
 - JVM, 22–23
 - language features, 19–22
 - packages enhancements, 22–23
 - ReentrantLock, 241
- Java 6 (Java SE 6)**
 - annotations, 50
 - Biased locking, 242
 - enumerations, 49–50
 - JVM, 23–25
- Java 7 (Java SE 7)**
 - Adaptive spinning, 243
 - annotations, 50
 - enumerations, 49–50
 - JVM, 26–30
 - language features, 25–26
 - NUMA architectures, 28–30
 - underscore (`_`), 51
- Java 8 (Java SE 8)**
 - annotations, 50–51
 - generics, 51
 - interface types, 51–52
 - JVM, 31–32
- language features, 30
- meta-annotations, 51
- StampedLock, 241–242
- “target-type” inference, 51
- underscore (`_`), 51
- Java 9 (Java SE 9), 32–34**
 - MethodHandles, 52–54
 - VarHandles, 43–44, 52–54
- Java 10 (Java SE 10), 34–35, 52–54**
- Java 11 (Java SE 11), 35–37**
 - modular JDK, 78, 82–83
 - unified logging, 113
- Java 12 (Java SE 12), 37**
 - modular JDK, 78, 82–83
 - switch expressions, 44, 55–56
- Java 13 (Java SE 13), 37–38**
 - modular JDK, 78, 82–83
 - switch expressions, 44, 55–56
 - yield keyword, 37
- Java 14 (Java SE 14), 38**
 - modular JDK, 78, 82–83
 - records, 57
 - switch expressions, 44, 55–56
- Java 15 (Java SE 15), 38–39**
 - modular JDK, 78, 82–83
 - records, 57
 - sealed classes, 38–39, 44, 56–57
- Java 16 (Java SE 16), 39–40**
 - modular JDK, 78, 82–83
 - records, 57
 - sealed classes, 38–39, 44, 56–57
- Java 17 (Java SE 17)**
 - deprecations, 42
 - encapsulation, 40–41
 - JDK, 40–41
 - JVM, 40
 - language features, 41
 - library enhancements, 41

- modular services, 78
 - example of, 80–81
 - implementing, 81–83
 - service consumers, 79–83
 - service providers, 79, 82–83
- removals, 42
- sealed classes, 38–39, 44, 56–57
- security, 40–41
- unified logging, 113
- Java Archive (JAR), JAR Hell versioning problem, 83–91**
- Java Development Kit (JDK), 1**
 - bytecode, 2
 - Java 17 (Java SE 17), 40–41
 - JRE, 2
 - modular JDK, 78, 82–83
- Java Executor Service, 261**
- Java Microbenchmark Harness (JMH)**
 - @Benchmark, 171
 - @BenchmarkMode, 171
 - @Fork, 171
 - @Measurement, 168–171
 - @OperationsPerInvocation, 169–171
 - @OutputTimeUnit, 171
 - @Setup, 171
 - @State, 171
 - @Teardown, 171
 - @Warmup, 168–171
 - annotations, 169–172
 - benchmarking, 172–174
 - benchmarking modes, 170
 - features of, 165
 - loop optimizations, 169
 - Maven, 166–170
 - measurement phase, 168–169
 - perfasm, 174
 - profilers, 170–171
 - profiling benchmarks, 174
- running microbenchmarks, 166–168
 - warm-up phase, 168–169
- writing/building microbenchmarks, 166–168
- Java Memory Model (JMM), thread synchronization, 237**
- Java Object Layout (JOL), 59–62**
- Java on Truffle, 292**
- Java Platform Module Service (JPMS), 84–85. See also modules**
- Java Runtime Environment (JRE), 2**
- Java Virtual Machine (JVM), 1**
 - bootstrapping, start-up performance, 275
 - future of, 336
 - Java 5 (J2SE 5.0), 22–23
 - Java 6 (Java SE 6), 23–25
 - Java 7 (Java SE 7), 26–30
 - Java 8 (Java SE 8), 31–32
 - Java 9 (Java SE 9), 32–34
 - Java 10 (Java SE 10), 34–35
 - Java 11 (Java SE 11), 35–36
 - Java 12 (Java SE 12), 37
 - Java 13 (Java SE 13), 37–38
 - Java 14 (Java SE 14), 38
 - Java 15 (Java SE 15), 38–39
 - Java 16 (Java SE 16), 39–40
 - Java 17 (Java SE 17), 40
 - language API, 333–334
 - memory management, benchmarking, 161–164
 - native libraries, 333–334
 - optimizing, serverless operations, 296–297
 - parameter tuning, footprint, performance engineering, 122
 - performance engineering, 153–154, 336
 - runtime environments, 153–154
 - serverless operations, 296–297

Shenandoah GC, 37

start-up performance, optimizing, JVM bootstrapping, 275

unified logging

- asynchronous logging integration, 111
- benchmarking, 108
- decorators, 104–105
- identifying missing information, 102
- infrastructure of, 100
- JDK 11, 113
- JDK 17, 113
- levels, 103–104
- log tags, 101
- managing systems, 109
- optimizing systems, 109
- outputs, 105–106
- performance, 108
- performance metrics, 101
- specific tags, 102
- tools/techniques, 108
- usage examples, 107–108

Jdepscan, 94–95

Jdeps, 93–94

JDK (Java Development Kit), 1

- bytecode, 2
- Java 17 (Java SE 17), 40–41
- JRE, 2
- modular JDK, 78, 82–83

jigsaw layers, 83–91

JIT (Just-in-Time) compiler, 3, 276

- OSR, 5
- state management, 283–285

Jlink, 96

JMH (Java Microbenchmark Harness)

- @Benchmark, 171
- @BenchmarkMode, 171
- @Fork, 171
- @Measurement, 168–171

@OperationsPerInvocation, 169–171

@OutputTimeUnit, 171

@Setup, 171

@State, 171

@Teardown, 171

@Warmup, 168–171

annotations, 169–172

benchmarking, 172–174

benchmarking modes, 170

features of, 165

loop optimizations, 169

Maven, 166–170

measurement phase, 168–169

perfasm, 174

profilers, 170–171

profiling benchmarks, 174

running microbenchmarks, 166–168

warm-up phase, 168–169

writing/building microbenchmarks, 166–168

JMM (Java Memory Model), thread synchronization, 237

Jmod, 95

JOL (Java Object Layout), 59–62

JPMS (Java Platform Module Service), 84–85. *See also* modules

JRE (Java Runtime Environment), 2

JShell, 32–34

Just-in-Time (JIT) compiler, 3, 276

- OSR, 5
- state management, 283–285

JVM (Java Virtual Machine), 1

- bootstrapping, start-up performance, 275
- future of, 336
- Java 5 (J2SE 5.0), 22–23
- Java 6 (Java SE 6), 23–25
- Java 7 (Java SE 7), 26–30
- Java 8 (Java SE 8), 31–32

Java 9 (Java SE 9), 32–34
 Java 10 (Java SE 10), 34–35
 Java 11 (Java SE 11), 35–36
 Java 12 (Java SE 12), 37
 Java 13 (Java SE 13), 37–38
 Java 14 (Java SE 14), 38
 Java 15 (Java SE 15), 38–39
 Java 16 (Java SE 16), 39–40
 Java 17 (Java SE 17), 40
 language API, 333–334
 memory management, benchmarking,
 161–164
 native libraries, 333–334
 optimizing, serverless operations,
 296–297
 parameter tuning, footprint,
 performance engineering, 122
 performance engineering,
 153–154, 336
 runtime environments, 153–154
 serverless operations, 296–297
 Shenandoah GC, 37
 start-up performance, optimizing, JVM
 bootstrapping, 275
 unified logging
 asynchronous logging integration, 111
 benchmarking, 108
 decorators, 104–105
 identifying missing information, 102
 infrastructure of, 100
 JDK 11, 113
 JDK 17, 113
 levels, 103–104
 log tags, 101
 managing systems, 109
 optimizing systems, 109
 outputs, 105–106
 performance, 108
 performance metrics, 101

specific tags, 102
 tools/techniques, 108
 usage examples, 107–108

K

klasses, 58–61

Kotlin, Project Valhalla comparisons, 68

L

language API, JVM, 333–334

**language design, exotic hardware,
 313–314**

Last-Level Caches (LLC), 137

latency, 123, 188

layers

 jigsaw layers, 83–91
 system stack layers, 151

LDS (Live Data Sets), 216–217

levels, unified logging, 103–104

libraries

 interactions, performance engineering,
 152–153
 Java 17 (Java SE 17), 41

**Lightweight Java Game Library (LWJGL),
 313–317**

literal strings, 221–223

literals, 45

Live Data Sets (LDS), 216–217

LLC (Last-Level Caches), 137

loading

 bytecode, 275–276
 classes, 9–10, 276

local traffic, NUMA, 143

lock coarsening, 242

lock elision, 242

log tags, 101

Log4j2, 110

Logback, 110

logging

asynchronous logging, 111–113
 unified logging
 asynchronous logging integration, 111
 benchmarking, 108
 decorators, 104–105
 identifying missing information, 102
 infrastructure of, 100
 JDK 11, 113
 JDK 17, 113
 levels, 103–104
 log tags, 101
 managing systems, 109
 need for, 99–100
 optimizing systems, 109
 outputs, 105–106
 performance, 108
 performance metrics, 101
 specific tags, 102
 tools/techniques, 108
 usage examples, 107–108

loop optimizations, benchmarking, 169**lots of threads, GC, 18****LWJGL (Lightweight Java Game Library), 313–317**

M**macOS/AArch64 port, 40****managing**

backpressure, asynchronous logging, 111
 footprint, performance engineering, 120
 heaps
 G1 GC, 184–188
 off-heap forwarding tables, 201
 ZGC, 201
 JVM memory, benchmarking, 161–164
 memory, optimizing, 217
 state, 278–280

AOT compilation, 283–285

benefits of, 281

JIT compilation, 283–285

unified logging systems, 109

Mark-Sweep-Compacting (MSC), 16, 22**marking thresholds, G1 GC, 196–197****Maven, microbenchmarking, 166–170****Mean Time Between Failure (MTBF), 124–126****Mean Time to Recovery (MTTR), 124–126****@Measurement, 168–171****measurement phase, benchmarking, 168–169****megamorphic call sites, 12****memory****access**

in multiprocessor systems, 141

optimizing with NUMA-aware GC, 181–183

performance, Project Valhalla, 66

DDR memory, 137–138

FFM API, 309, 330–333

footprint, 119

hierarchies, 136–138

interleaved memory, NUMA, 143

JMM, thread synchronization, 237

JVM memory management, benchmarking, 161–164

mapping, shared archive files, 282

memory models, 131–136

NMT, monitoring non-heap memory, 120–122

non-heap memory, monitoring with NMT, 120–122

NUMA, 141, 145

architecture of, 143

ccNUMA, 144

components of, 142–143

cross-traffic, 143

fabric architectures, 144
GC (Garbage Collector), 181–183
 interleaved memory, 143
 Java 7 (Java SE 7), 28–30
 local traffic, 143
 nodes in modern systems, 142
 NUMA-aware memory allocator, 38
 object memory layouts, 59–62
 optimizing, 217
 value classes, 63–66

mesh fabric architectures, Intel, 144

meta-annotations, 51

Metaspace, 302–304

MethodHandles, 52–54

microbenchmarking

- JMH, 172–174
- @Benchmark, 171
- @BenchmarkMode, 171
- @Fork, 171
- @Measurement, 168–169, 171
- @OperationsPerInvocation, 169, 171
- @OutputTimeUnit, 171
- @Setup, 171
- @State, 171
- @Teardown, 171
- @Warmup, 168–169, 171
- annotations, 169, 171–172
- benchmarking modes, 170
- features of, 165
- loop optimizations, 169
- Maven, 166–170
- measurement phase, 168–169
- perfasm, 174
- profilers, 170–171
- profiling benchmarks, 174
- running microbenchmarks, 166–168
- warm-up phase, 168–169

writing/building microbenchmarks, 166–168
 Maven, 166–170
 warm-up phase, 168–169

mixed-mode execution, Hotspot VM, 3

modifying, modules, 74–75

modular JDK (Java Development Kit), 78, 82–83

modular services, 78

- example of, 80–81
- implementing, 81–83
- service consumers, 79–83
- service providers, 79, 82–83

ModuleLayer, 84–85

modules. See also JPMs

- compiling, 72–76
- defined, 70
- example of, 71–72
- Jdeprscan, 94–95
- Jdeps, 93–94
- Jlink, 96
- Jmod, 95
- jigsaw layers, 83–91
- modifying, 74–75
- OSGi comparisons, 91–93
- performance, 97
- relationships between, 76
- role of, 70
- running, 72–76
- service consumers, 79–83
- service providers, 79, 82–83
- updating, 74–75
- use-case diagrams, 77

monitor enter operations, 243–245

monitor exit operations, 243–245

monitor locks

- contented locks, 239–241
- Adaptive spinning (Java 7), 243

- benchmarking, 248–251
 - call tree analysis, 251–254
 - flamegraph analysis, 249–251
 - lock coarsening, 242
 - lock elision, 242
 - monitor enter operations, 243–245
 - monitor exit operations, 243–245
 - PAUSE instructions, 258–259
 - performance engineering, 245–259
 - ReentrantLock (Java 5), 241
 - spin-loop hints, 258–259
 - spin-wait hints, 257–259
 - StampedLock (Java 8), 241–242
 - deflated locks, 238–239
 - inflated locks, 239–241
 - role of, 238
 - types of, 238–241
 - uncontended locks, 238–239, 242
 - monomorphic call sites, 12**
 - mpstat tool, 156
 - MSC (Mark-Sweep-Compacting), 16, 22**
 - MTBF (Mean Time Between Failure), 124–126**
 - MTTR (Mean Time to Recovery), 124–126**
 - multidimensional arrays, 49**
 - multiprocessor systems**
 - memory access, 141
 - NUMA, 141, 145
 - architecture of, 143
 - ccNUMA, 144
 - components of, 142–143
 - cross-traffic, 143
 - fabric architectures, 144
 - hops, 28–30
 - interleaved memory, 143
 - local traffic, 143
 - nodes in modern systems, 142
 - multithreading**
 - SMT, concurrent hardware, 136
 - synchronizing threads, 236
 - contented locks, 239–259
 - happens-before relationships, 237
 - JMM, 237
 - monitor locks, 238–259
 - Mustang.** *See Java 6 (Java SE 6)*
 - mutation rates, regionalized heaps, 188**
-
- N**
- native image generation, GraalVM, 291**
 - native libraries, JVM (Java Virtual Machine), 333–334**
 - Native Memory Tracking (NMT), 120–122**
 - ndrange API, 335–336**
 - NetBeans IDE, compact strings, 229–236**
 - nmethod, 7–8**
 - NMT (Native Memory Tracking), 120–122**
 - non-heap memory, monitoring with NMT, 120–122**
 - non-method code heap, segmented code cache, 8**
 - non-profiled nmethod code heap, 8**
 - null, 45**
 - NUMA (Non-Uniform Memory Access), 141, 145**
 - architecture of, 143
 - ccNUMA, 144
 - components of, 142–143
 - cross-traffic, 143
 - fabric architectures, 144
 - GC (Garbage Collector), 181–183
 - hops, 28–30
 - interleaved memory, 143
 - Java 7 (Java SE 7), 28–30
 - local traffic, 143
 - nodes in modern systems, 142
 - NUMA-aware memory allocator, 38

O**objects**

- components of, 58–59
- immutable objects, Project Valhalla, 63
- klasses, 58–61
- memory layouts, 59–62
- value objects, Project Valhalla, 66

off-heap forwarding tables, ZGC, 201**OLAP (Online Analytical Processing), 214–215****old-generation collections, 16****OLTP (Online Transaction Processing) systems**

- operational stores, 215
- performance engineering, 149–151

On-Stack Replacement (OSR), 5**OpenCL, 308, 314, 317–321****OpenJDK**

- concurrent algorithms, 14
- CRIU, 292–295
- GC
 - future trends, 210–212
 - G1 GC, 178–179, 184–197
 - NUMA-aware GC, 181–183
 - performance evaluations, 212–216
 - PLAB, 180–181
 - TLAB, 179–180
 - ZGC, 178–179, 197–210
- GraalVM, 291
- incremental compacting algorithms, 14
- PLAB, 180–181
- Project CRaC, 292–295
- STW algorithms, 14
- thread-local handshakes, 14
- TLAB, 179–180
- ultra-low-pause-time collectors, 14

Open Services Gateway initiative (OSGi), 91–93**operational stores (OLTP), 215****@OperationsPerInvocation, 169–171****optimizing**

- data structures, footprint, 122
- G1 GC, 188–197
- JVM, serverless operations, 296–297
- memory, 217
- performance-critical methods, 3–5
- runtime performance, 219
 - strings, 220–236
 - threads, 236–270
- start-up performance, 274–275
- strings, 220
 - compact strings, 227–236
 - concatenating, 224–227
 - deduplicating (dedup), 223–224
 - indy-fication, 224–227
 - interned strings, 221–223
 - literal strings, 221–223
 - reducing footprint, 224–236
- threads, synchronization, 236
 - contented locks, 239–259
 - happens-before relationships, 237
 - JMM, 237
 - monitor locks, 238–259
 - unified logging systems, 109
 - warm-up performance, 274

OS (Operating Systems), performance engineering, 155–156**OSGi (Open Services Gateway initiative), 91–93****OSR (On-Stack Replacement), 5****outputs, unified logging, 105–106****@OutputTimeUnit, 171****P****parallel GC threads, 16–18****parallelism, virtual threads, 269–270**

pattern matching, instanceof operator, 38

PAUSE instructions, contended locks, 258–259

pause responsiveness, G1 GC, 189–193

pauses, GC, 17

pause-time predictability, heap management, 184–185

perfasm, profiling JMH benchmarks, 174

performance

- asynchronous logging, 112–113
- availability
 - MTBF, 124–126
 - MTTR, 124–126
- cloud computing, 311
- concurrent computing
 - atomicity, 139–141
 - barriers, 139
 - core utilization, 136
 - fences, 139
 - happens-before relationships, 139–141
 - memory access in multiprocessor systems, 141
 - memory hierarchies, 136–138
 - memory models, 139–141
 - NUMA, 141–145
 - processors, 136–138
 - volatiles, 139
- engineering. *See separate entry*
- evaluating, 117
- exotic hardware, 311
- footprint
 - adaptive sizing policies, 123
 - code footprint, 119
 - code refactoring, 122
 - data structure optimization, 122
 - JVM parameter tuning, 122
 - managing, 120
 - memory footprint, 119
 - mitigating issues, 122–123
- monitoring non-heap memory with NMT, 120–122
- NMT, 120–122
- physical resources, 120
- G1 GC, 188–197, 217
- GC, 212–216
- hardware, 128
- caches, 132
- concurrent hardware, 136–138
- hardware-aware programming, 132
- memory hierarchies, 136–138
- processors, 136–138
- SMT, 136
- software dynamics, 129–131
- memory access performance, Project Valhalla, 66
- memory models, 131–132
 - atomicity, 139–141
 - thread dynamics, 133–136
- Metaspace, 304–306
- metrics, 118–128
- modules, 97
- performance-critical methods, optimizing, 3–5
- PermGen, 304–306
- PMU, 157
- processors, 136–138
- Project Leyden, 285–290
- Project Valhalla, 58–63
- QoS, defined, 117–118
- ramp-up performance, 298–300
 - application lifecycles, 279
 - defined, 273
 - serverless operations, 295–296
 - tasks (overview), 276–277
 - transitioning to steady-state, 281
- response times, 123, 127–128
 - 4–9s, 124–126
 - 5–9s, 125–126

- runtime performance, 219
- strings, 220–236
- threads, 236–270
- start-up performance, 274, 297
 - application lifecycles, 278
 - CDS, 281–282
 - containerized environments, 296–297
 - Hotspot VM, 300
 - JVM bootstrapping, 275
 - Metaspace, 302
 - PermGen, 302
 - serverless operations, 294–295
- SUT, 117
- thread synchronization, 236
 - contended locks, 239–259
 - happens-before relationships, 237
 - JMM, 237
 - monitor locks, 238–259
- throughput, 123–124
- unified logging, 101, 108
- UoW, 117
- warm-up performance
 - defined, 273
 - Hotspot VM, 302
 - Metaspace, 304–306
 - optimizing, 274
 - PermGen, 304–306
- ZGC, 217
- performance engineering, 3–5, 115–116**
 - availability
 - MTBF, 124–126
 - MTTR, 124–126
 - benchmarking, 158
 - harnesses, 164–165
 - iterative approach, 161
 - JMH, 165–174
 - JVM memory management, 161–164
 - metrics, 159
- microbenchmarking, 165–174
- process (overview), 159–161
- bottom-up methodology, 146–148
- concurrent computing
 - atomicity, 139–141
 - barriers, 139
 - core utilization, 136
 - fences, 139
 - happens-before relationships, 139–141
 - memory access in multiprocessor systems, 141
 - memory hierarchies, 136–138
 - memory models, 139–141
 - NUMA, 141–145
 - processors, 136–138
 - volatiles, 139
- containerized environments, 155
- contended locks, 245–259
- evaluating, 117
- experimental design, 146
- footprint
 - adaptive sizing policies, 123
 - code footprint, 119
 - code refactoring, 122
 - data structure optimization, 122
 - JVM parameter tuning, 122
 - managing, 120
 - memory footprint, 119
 - mitigating issues, 122–123
 - monitoring non-heap memory with NMT, 120–122
 - NMT, 120–122
 - physical resources, 120
- GC, 154
- hardware
 - caches, 132
 - concurrent hardware, 136–138
 - core utilization, 136

memory hierarchies, 136–138
 processors, 136–138
 SMT, 136
 hardware subsystems, 156–157
 hardware’s role in, 128
 hardware-aware programming, 132
 software dynamics, 129–131
 hypervisors, 156
 JVM, future of, 336
 memory models, 131–133
 atomicity, 139–141
 thread dynamics, 133–136
 methodologies, 145–150
 metrics, 118–128
 OLTP systems, 149–151
 OS, 155–156
 PMU, 157
 processors, 136–138
 QoS, defined, 117–118
 response times, 123, 127–128
 4–9s, 124–126
 5–9s, 125–126
 SoW, 149–151
 subsystems
 async-profiler, 157
 components of (overview), 152
 containerized environments, 155
 GC, 154
 hardware, 156–157
 host OS, 156
 hypervisors, 156
 interactions, 152–153
 Java application profilers, 157
 JVM, 153–154
 libraries, 152–153
 OS, 155–156
 PMU, 157
 runtime environments, 153–154
 system infrastructures, 152–153
 system profilers, 157
 system stack layers, 151
 SUT, 117
 system infrastructures, 152–153
 system profilers, 157
 throughput, 123–124
 top-down methodology, 148–158
 UoW, 117

Performance Monitoring Units (PMU), 157

PermGen (Permanent Generation), 304–306

phases, ZGC, 199–201

physical resources, footprint, 120

PLAB (Promotion-Local Allocation Buffers), 180–181

PMU (Performance Monitoring Units), 157

pointers (colored), ZGC, 197–198

polymorphic call sites, 11–13

“pool of strings”, 221

ports

- macOS/AArch64 port, 40
- windows/AArch64 port, 39

primitive classes, 65–66

primitive data types, 44–45

print compilation, Hotspot VM, 5–6

proactive GC (Garbage Collection), 208–209

processors

- fabric architectures, 144
- multiprocessor systems, memory access, 141
- performance, 136–138

profiled nmethod code heap, 8

profilers, JMH, 170–171, 174

Project Babylon, 336

Project CRaC, 292–295

Project Jigsaw, 32–34

- Project Leyden, 285–290**
- Project Lilliput, 237**
- Project Loom, 237, 266**
- Project Panama**
 - FFM API, 309, 330–333
 - vector API, 308, 327–330, 334–335
- Project Sumatra, 314, 321–324**
- Project Valhalla, 44, 237**
 - arrays, 60–62
 - C# comparisons, 67–68
 - classes, 65–66
 - command-line options, 67
 - experimental features, 67
 - generics, 64–66
 - Hotspot VM, 67
 - immutable objects, 63
 - JOL, 59–62
 - Kotlin comparisons, 68
 - memory access performance, 66
 - object memory layouts, 59–62
 - performance implications, 58–63
 - Scala comparisons, 68
 - use case scenarios, 67
 - value classes, 63–66
 - value objects, 66
- Promotion-Local Allocation Buffers (PLAB), 180–181**

- Q**
- Quality of Service (QoS), defined, 117–118**

- R**
- ramp-down performance, life cycle of applications, 278**
- ramp-up performance, 298–300**
 - applications, life cycle of, 278
 - defined, 273
 - serverless operations, 295–296
- tasks (overview), 276–277
- transitioning to steady-state, 281
- readable code, underscore (`_`), 51**
- reclamation triggers, 16**
- records (Java 16, Java 17), 39, 57**
- recovery, MTTR, 124–126**
- ReentrantLock (Java 5), 241**
- refactoring code, footprint, 122**
- reference types, 45–49**
- regionalized heaps, 184, 186–188**
- relationships between modules, 76**
- release cadence, Java, 34**
- reliability, asynchronous logging, 112**
- removals, Java 17 (Java SE 17), 42**
- requirements gathering (benchmarking performance), 160**
- resource contention, exotic hardware, 311**
- response times, performance engineering, 123, 127–128**
 - 4–9s, 124–126
 - 5–9s, 125–126
 - STW pauses, 126–128
- responsiveness**
 - G1 GC, pause responsiveness, 189–193
 - regionalized heaps, 188
- restore/checkpoint functionality, CRIU, 291–294**
- running modules, 72–76**
- runtime environments, performance engineering, 153–154**
- runtime performance, strings, 219–220**
 - compact strings, 227–236
 - concatenating, 224–227
 - deduplicating (dedup), 223–224
 - indy-fication, 224–227
 - interned strings, 221–223
 - literal strings, 221–223
 - reducing footprint, 224–236

S

-
- Scala, Project Valhalla comparisons, 68**
- scalability**
- containerized environments, 297–298
 - G1 GC, 188
 - threads
 - CompletableFuture frameworks, 264–265
 - ForkJoinPool frameworks, 261–264
 - Java Executor Service, 261
 - thread pools, 261
 - thread-per-request model, 261–266
 - thread-per-task model, 259–260
 - virtual threads, 265–270**
- scavenge algorithm, 16**
- sealed classes (Java 15, Java 17), 38–39, 44, 56–57**
- security, Java 17 (Java SE 17), 40–41**
- Segmented CodeCache, 7–8, 300–302**
- server compiler (C2), 7**
- serverless operations**
- JVM optimization, 296–297
 - ramp-up performance, 295–296
 - start-up performance, 294–295
- service consumers, 79–80, 82–83**
- Service Level Agreements (SLA), 117–118, 124–125**
- Service Level Indicators (SLI), 117–118, 123**
- Service Level Objectives (SLO), 117–118, 123**
- service providers, 79, 82–83**
- ServiceLoader API, 81–83**
- @Setup, 171**
- shared archive files, 282–283**
- Shenandoah GC, 16, 37**
- Simultaneous Multithreading (SMT), 136**
- sizing policies, footprint, 123**
- SLA (Service Level Agreements), 117–118, 124–125**
- SLC (System-Level Caches), 137**
- SLI (Service Level Indicators), 117–118, 123**
- SLO (Service Level Objectives), 117–118, 123**
- slow-debug, 99**
- SMT (Simultaneous Multithreading), 136**
- software**
- concurrency, hardware interplay, 131
 - engineering, layers of, 116–117
 - hardware dynamics in performance, 129–131
- SoW (Statements of Work), 149–151**
- Spark (Apache)**
- GC collection performance, 214
 - Vector API, 334
- specific tags, 102**
- spin-loop hints, contended locks, 258–259**
- spin-wait hints, contended locks, 257–259**
- StampedLock (Java 8), 241–242**
- start-up performance, 297**
- applications, life cycle of, 278
 - CDS, 282
 - containerized environments, 297–298
 - Hotspot VM, 300–302
 - JVM bootstrapping, 275
 - Metaspace, 304–306
 - optimizing, 274–275
 - PermGen, 304–306
 - serverless operations, 294–295
- @State, 171**
- state management**
- AOT compilation, 283–285
 - benefits of, 281
 - JIT compilation, 283–285
 - start-up performance, 278–280
- Statements of Work (SoW), 149–151**
- statically type-checked language, Java as, 43**

- steady-state phase**
 - applications, life cycle of, 278
 - GraalVM, 290–291
 - Hotspot VM, 301
 - Metaspace, 304–306
 - PermGen, 304–306
 - transitioning ramp-up performance to, 281
 - stealing tasks, GC, 17**
 - stopping, applications, 278**
 - Stop the World (STW) algorithms, 14**
 - Stop the World (STW) pauses, response times, 126–128**
 - strings, 220**
 - compact strings, 227–236
 - concatenating, 224–227
 - deduplicating (dedup), 223–224
 - indy-fication, 224–227
 - interned strings, 221–223
 - literal strings, 221–223
 - NetBeans IDE, 229–236
 - “pool of strings”, 221
 - reducing footprint, 224–236
 - storing, 221–223
 - visualizing representations, 228
 - strongly typed language, Java as, 43**
 - STW (Stop the World) algorithms, 14**
 - STW (Stop the World) pauses, response times, 126–128**
 - subsystems, performance engineering**
 - async-profiler, 157
 - components of (overview), 152
 - containerized environments, 155
 - GC, 154
 - hardware, 156–157
 - host OS, 156
 - hypervisors, 156
 - interactions, 152–153
 - Java application profilers, 157
 - JVM, 153–154**
 - libraries, 152–153**
 - OS, 155–156**
 - PMU, 157**
 - runtime environments, 153–154**
 - system infrastructures, 152–153**
 - system profilers, 157**
 - system stack layers, 151**
 - SUT (System Under Test), 117**
 - sweepers, 8**
 - switch expressions (Java 12, Java 14), 37, 44, 55–56**
 - synchronizing threads, 236**
 - happens-before relationships, 237
 - JMM, 237
 - monitor locks
 - contentended locks, 239–259
 - deflated locks, 238–239
 - inflated locks, 239–241
 - role of, 238
 - types of, 238–241
 - uncontended locks, 238–239
 - sysstat tool, 156**
 - system infrastructures, performance engineering, 152–153**
 - System-Level Caches (SLC), 137**
 - system profilers, performance engineering, 157**
 - system stacks, layers of, 151**
 - System Under Test (SUT), 117**
-
- T**
-
- tags, unified logging, 101–102**
 - tail latency, regionalized heaps, 188**
 - “target-type” inference, 51**
 - task queues, 17**
 - task stealing, GC, 17**
 - @Teardown, 171**
 - TemplateTable, Hotspot VM, 3**

test planning/development, benchmarking performance, 160

text blocks, 37–38

Thread-Local Allocation Buffers (TLAB), 179–180

threads

- CompletableFuture frameworks, 264–265
- concurrent computing
 - CompletableFuture frameworks, 264–265
 - ForkJoinPool frameworks, 261–264
 - Java Executor Service, 261
 - thread pools, 261
 - thread-per-request model, 259–260
 - thread-per-task model, 261–266
 - virtual threads, 266–270
- concurrent thread-stack processing, 39–40
- ForkJoinPool frameworks, 261–264
- GC, 18
- memory models, 133–136
- monitor locks
 - contended locks, 239–259
 - deflated locks, 238–239
 - inflated locks, 239–241
 - role of, 238
 - types of, 238–241
 - uncontended locks, 238–239
- scalability
 - CompletableFuture frameworks, 264–265
 - ForkJoinPool frameworks, 261–264
 - Java Executor Service, 261
 - thread pools, 261
 - thread-per-request model, 261–266
 - thread-per-task model, 259–260
 - virtual threads, 265–270

 synchronizing, 236

 contended locks, 239–259

 happens-before relationships, 237

 JMM, 237

 monitor locks, 238–259

 thread pools, 261

 thread-local handshakes, 14, 198–199

 TLAB, 179–180

 virtual threads

- API integration, 267
- carriers, 267
- continuations, 270
- example of, 267–269
- parallelism, 269–270
- Project Loom, 266

throughput, performance engineering, 123–124

tiered compilation, Hotspot VM, 6–7

time-based GC, ZGC, 204–205

TLAB (Thread-Local Allocation Buffers), 179–180

toolchains, exotic hardware, 313–314

top-down methodology, performance engineering, 148–158

TornadoVM, 308, 314, 324–327, 336

U

ultra-low-pause-time collectors, 14

uncontented locks, 238–239, 242

- Biased locking (Java 6), 242

underscore (_), code readability, 51

unified logging

- asynchronous logging integration, 111
- benchmarking, 108
- decorators, 104–105
- identifying missing information, 102
- infrastructure of, 100
- JDK 11, 113
- JDK 17, 113

levels, 103–104
 log tags, 101
 managing systems, 109
 need for, 99–100
 optimizing systems, 109
 outputs, 105–106
 performance, 108
 performance metrics, 101
 specific tags, 102
 tools/techniques, 108
 usage examples, 107–108

Unit of Work (UoW), 117

unloading/loading classes, Hotspot VM
 deoptimization, 9–10
 updating, modules, 74–75
 use case scenarios, Project Valhalla, 67
 use-case diagrams, modules, 77

V

validation, benchmarking performance, 160
 value classes, 63–66
 value objects, Project Valhalla, 66
VarHandles, 43–44, 52–54
 vector API, Project Panama, 308, 327–330,
 334–335
 verifying bytecode, optimizing start-up
 performance, 275–276
 versioning, JAR Hell versioning problem,
 83–91
virtual threads, 265
 API integration, 267
 carriers, 267
 continuations, 270
 example of, 267–269
 parallelism, 269–270
 Project Loom, 266
virtualized hardware. See exotic hardware
vmstat utility, 155–156
volatiles, concurrent computing, 139

W

warming caches, 276
@Warmup, 168–169, 171
**warm-up-based GC (Garbage Collection),
 205–206**
warm-up performance
 defined, 273
 Hotspot VM, 300–303
 Metaspace, 304–306
 optimizing, 274
 PermGen, 304–306
**warm-up phase, benchmarking,
 168–169**
weak generational hypothesis, 14–16
windows/AArch64 port, 39

X - Y - Z

yield keyword (Java 13), 37
young collections, 14–16
**Z Garbage Collector (ZGC), 16, 35–38,
 178–179**
 advancements, 209–210
 allocation-stall-based GC, 207
 colored pointers, 197–198
 concurrent computing, 198–200
 concurrent thread-stack processing,
 39–40
 high allocation rate-based GC,
 206–207
 high-usage-based GC, 207–208
 performance, 217
 phases, 199–201
 proactive GC, 208–209
 thread-local handshakes, 198–199
 time-based GC, 204–205
 triggering cycles, 204–209
 warm-up-based GC, 205–206
 ZPages, 198, 202–204

This page intentionally left blank

Goodbye, database complexity. Hello, simplicity.

Reduce complexity and cost with the one database for all your workloads. Cloud native, converged, and automated, Oracle Autonomous Database eliminates manual maintenance hassles and delivers top performance, security, and automatic scaling for your changing needs. It also supports the full development lifecycle for modern applications.

Get simplicity in the cloud or on-premises.

Oracle Autonomous Database

Learn more at

oracle.com/autonomous-database





The #1 language
for today's tech trends

www.oracle.com/java



Take your skills to the next level with *Java Magazine*

Written for developers by developers,
new articles are posted every two weeks,
and there's a fresh quiz every Tuesday!

Read all the articles and subscribe
to the newsletter at no cost.
oracle.com/javamagazine



Oracle VM VirtualBox

Oracle's open source, cross-platform, desktop hypervisor helps deliver code faster

Deploy your applications quickly and securely, on-premises and to the cloud, with more productivity and simpler operations, at a lower cost.

- Cloud integration
- Runs on any desktop
- Full VM encryption

oracle.com/virtualbox





Register Your Product

at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

* Offers subject to change.

** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

Connect with InformIT—Visit informit.com/community

