Now let's look at the directory structure for `com.house.bricks`:

```
src
└── com.house.bricks
    ├── com
    │   └── house
    │       └── bricks
    │           └── Story.java
    └── module-info.java

com.house.bricks:
   module-info.java:

module com.house.bricks {
    exports com.house.bricks;
}

com/house/bricks/Story.java:

package com.house.bricks;

public class Story {
    public static int count(int level) {
        return level * 18000;
    }
}
```

Compilation and Run Details

We compile the `com.house.bricks` module first:

```
$ javac -d mods/com.house.bricks src/com.house.bricks/module-info.java src/com.house.bricks/com/
house/bricks/Story.java
```

Next, we compile the `com.house.brickhouse` module:

```
$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

Now we run the `House1` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House1
```

Output:

```
My single-level house will need 18000 bricks
```

Then we run the `House2` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House2
```

Output:

```
My two-level house will need 36000 bricks
```

Introducing a New Module

Now, let's expand our project by introducing a new module that provides various types of bricks. We'll call this module `com.house.bricktypes`, and it will include different classes for different types of bricks. Here's the new directory structure for the `com.house.bricktypes` module:

```
src
└── com.house.bricktypes
    ├── com
    │   └── house
    │       └── bricktypes
    │           ├── ClayBrick.java
    │           └── ConcreteBrick.java
    └── module-info.java

com.house.bricktypes:
    module-info.java:

module com.house.bricktypes {
    exports com.house.bricktypes;
}
```

The `ClayBrick.java` and `ConcreteBrick.java` classes will define the properties and methods for their respective brick types.

`ClayBrick.java`:

```
package com.house.bricktypes;

public class ClayBrick {
    public static int getBricksPerSquareMeter() {
        return 60;
    }
}
```

ConcreteBrick.java:

```
package com.house.bricktypes;

public class ConcreteBrick {
    public static int getBricksPerSquareMeter() {
        return 50;
    }
}
```

With the new module in place, we need to update our existing modules to make use of these new brick types. Let's start by updating the `module-info.java` file in the `com.house.brickhouse` module:

```
module com.house.brickhouse {
    requires com.house.bricks;
    requires com.house.bricktypes;
    exports com.house.brickhouse;
}
```

We modify the `House1.java` and `House2.java` files to use the new brick types.

House1.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ClayBrick;

public class House1 {
    public static void main(String[] args) {
        int bricksPerSquareMeter = ClayBrick.getBricksPerSquareMeter();
        System.out.println("My single-level house will need "
                           + Story.count(1, bricksPerSquareMeter) + " clay bricks");
    }
}
```

House2.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ConcreteBrick;

public class House2 {
```

```
    public static void main(String[] args) {
        int bricksPerSquareMeter = ConcreteBrick.getBricksPerSquareMeter();
        System.out.println("My two-level house will need "
                           + Story.count(2, bricksPerSquareMeter) + " concrete bricks");
    }
}
```

By making these changes, we're allowing our `House1` and `House2` classes to use different types of bricks, which adds more flexibility to our program. Let's now update the `Story.java` class in the `com.house.bricks` module to accept the bricks per square meter:

```
package com.house.bricks;

public class Story {
    public static int count(int level, int bricksPerSquareMeter) {
        return level * bricksPerSquareMeter * 300;
    }
}
```

Now that we've updated our modules, let's compile and run them to see the changes in action:

- Create a new `mods` directory for the `com.house.bricktypes` module:

```
$ mkdir mods/com.house.bricktypes
```

- Compile the `com.house.bricktypes` module:

```
$ javac -d mods/com.house.bricktypes
src/com.house.bricktypes/module-info.java
src/com.house.bricktypes/com/house/bricktypes/*.java
```

- Recompile the `com.house.bricks` and `com.house.brickhouse` modules:

```
$ javac --module-path mods -d mods/com.house.bricks
src/com.house.bricks/module-info.java src/com.house.bricks/com/house/bricks/Story.java

$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

With these updates, our program is now more versatile and can handle different types of bricks. This is just one example of how the modular system in Java can make our code more flexible and maintainable.

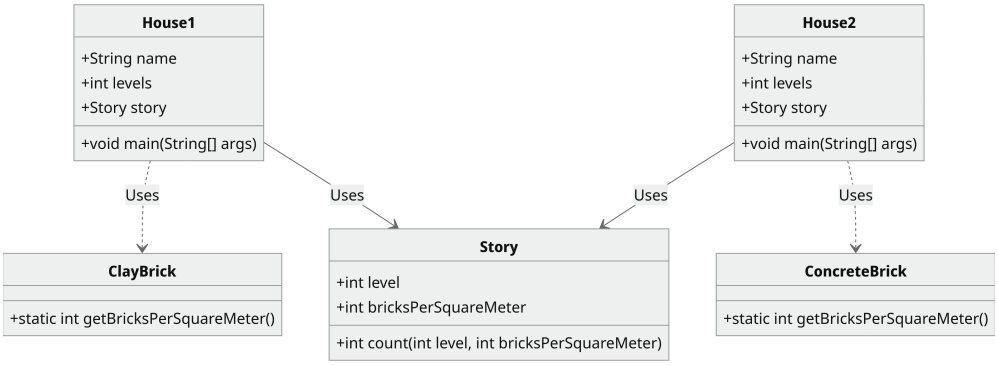

Figure 3.1 Class Diagram to Show the Relationships Between Modules

Let's now visualize these relationships with a class diagram. Figure 3.1 includes the new module `com.house.bricktypes`, and the arrows represent "Uses" relationships. `House1` uses `Story` and `ClayBrick`, whereas `House2` uses `Story` and `ConcreteBrick`. As a result, instances of `House1` and `House2` will contain references to instances of `Story` and either `ClayBrick` or `ConcreteBrick`, respectively. They use these references to interact with the methods and attributes of the `Story`, `ClayBrick`, and `ConcreteBrick` classes. Here are more details:

- `House1` and `House2`: These classes represent two different types of houses. Both classes have the following attributes:
  - `name`: A string representing the name of the house.
  - `levels`: An integer representing the number of levels in the house.
  - `story`: An instance of the `Story` class representing a level of the house.
  - `main(String[] args)`: The entry method for the class, which acts as the initial kickstarter for the application's execution.
- `Story`: This class represents a level in a house. It has the following attributes:
  - `level`: An integer representing the level number.
  - `bricksPerSquareMeter`: An integer representing the number of bricks per square meter for the level.
  - `count(int level, int bricksPerSquareMeter)`: A method that calculates the total number of bricks required for a given level and bricks per square meter.
- `ClayBrick` and `ConcreteBrick`: These classes represent two different types of bricks. Both classes have the following attributes:
  - `getBricksPerSquareMeter()`: A static method that returns the number of bricks per square meter. This method is called by the houses to obtain the value needed for calculations in the `Story` class.

Next, let's look at the use-case diagram of the Brick House Construction system with the House Owner as the actor and Clay Brick and Concrete Brick as the systems (Figure 3.2). This diagram illustrates how the House Owner interacts with the system to calculate the number of bricks required for different types of houses and choose the type of bricks for the construction.

Here's more information on the elements of the use-case diagram:

- **House Owner:** This is the actor who wants to build a house. The House Owner interacts with the Brick House Construction system in the following ways:
  - **Calculate Bricks for House 1:** The House Owner uses the system to calculate the number of bricks required to build House 1.
  - **Calculate Bricks for House 2:** The House Owner uses the system to calculate the number of bricks required to build House 2.
  - **Choose Brick Type:** The House Owner uses the system to select the type of bricks to be used for the construction.
- **Brick House Construction:** This system helps the House Owner in the construction process. It provides the following use cases:
  - **Calculate Bricks for House 1:** This use case calculates the number of bricks required for House 1. It interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Calculate Bricks for House 2:** This use case calculates the number of bricks required for House 2. It also interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Choose Brick Type:** This use case allows the House Owner to choose the type of bricks for the construction.
- **Clay Brick and Concrete Brick:** These systems provide the data (e.g., size, cost) to the Brick House Construction system that is needed to calculate the number of bricks required for the construction of the houses.

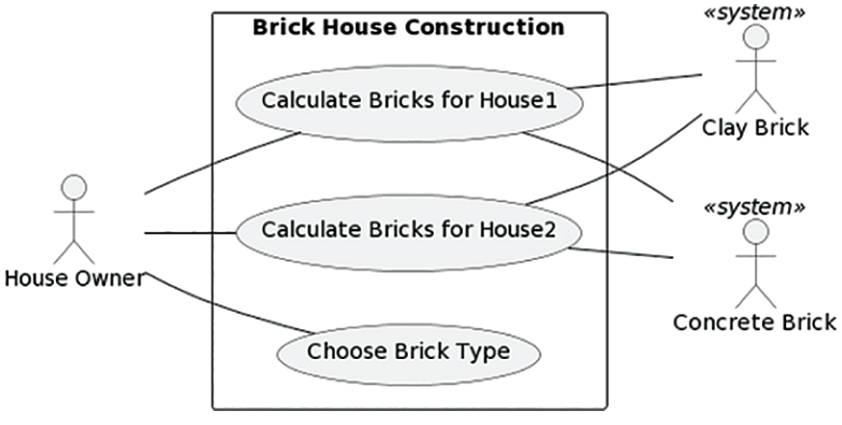

Figure 3.2 Use-Case Diagram of Brick House Construction

From Monolithic to Modular: The Evolution of the JDK

Before the introduction of the modular JDK, the bloating of the JDK led to overly complex and difficult-to-read applications. In particular, complex dependencies and cross-dependencies made it difficult to maintain and extend applications. JAR (Java Archives) hell (i.e., problems related to loading classes in Java) arose due to both the lack of simplicity and JARs' lack of awareness about the classes they contained.

The sheer footprint of the JDK also posed a challenge, particularly for smaller devices or other situations where the entire monolithic JDK wasn't needed. The modular JDK came to the rescue, transforming the JDK landscape.

Continuing the Evolution: Modular JDK in JDK 11 and Beyond

The Java Platform Module System (JPMS) was first introduced in JDK 9, and its evolution has continued in subsequent releases. JDK 11, the first long-term support (LTS) release after JDK 8, further refined the modular Java platform. Some of the notable improvements and changes made in JDK 11 are summarized here:

- **Removal of deprecated modules:** Some Java Enterprise Edition (EE) and Common Object Request Broker Architecture (CORBA) modules that had been deprecated in JDK 9 were finally removed in JDK 11. This change promoted a leaner Java platform and reduced the maintenance burden.
- **Matured module system:** The JPMS has matured over time, benefiting from the feedback of developers and real-world usage. Newer JDK releases have addressed issues, improved performance, and optimized the module system's capabilities.
- **Refined APIs:** APIs and features have been refined in subsequent releases, providing a more consistent and coherent experience for developers using the module system.
- **Continued enhancements:** JDK 11 and subsequent releases have continued to enhance the module system—for example, by offering better diagnostic messages and error reporting, improved JVM performance, and other incremental improvements that benefit developers.

Implementing Modular Services with JDK 17

With the JDK's modular approach, we can enhance the concept of services (introduced in Java 1.6) by decoupling modules that provide the service interface from their provider module, eventually creating a fully decoupled consumer. To employ services, the type is usually declared as an interface or an abstract class, and the service providers need to be clearly identified in their modules, enabling them to be recognized as providers. Lastly, consumer modules are required to utilize those providers.

To better explain the decoupling that occurs, we'll use a step-by-step example to build a `BricksProvider` along with its providers and consumers.

Service Provider

A service provider is a module that implements a service interface and makes it available for other modules to consume. It is responsible for implementing the functionalities defined in the service interface. In our example, we'll create a module called `com.example.bricksprovider`, which will implement the `BrickHouse` interface and provide the service.

Creating the `com.example.bricksprovider` Module

First, we create a new directory called `bricksprovider`; inside it, we create the `com/example/bricksprovider` directory structure. Next, we create a `module-info.java` file in the `bricksprovider` directory with the following content:

```
module com.example.bricksprovider {
    requires com.example.brickhouse;
    provides com.example.brickhouse.BrickHouse with com.example.bricksprovider.BricksProvider;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and provides an implementation of the `BrickHouse` interface through the `com.example.bricksprovider.BricksProvider` class.

Now, we create the `BricksProvider.java` file inside the `com/example/bricksprovider` directory with the following content:

```
package com.example.bricksprovider;
import com.example.brickhouse.BrickHouse;

public class BricksProvider implements BrickHouse {
    @Override
    public void build() {
        System.out.println("Building a house with bricks...");
    }
}
```

Service Consumer

A service consumer is a module that uses a service provided by another module. It declares the service it requires in its `module-info.java` file using the `uses` keyword. The service consumer can then use the `ServiceLoader` API to discover and instantiate implementations of the required service.

Creating the `com.example.builder` Module

First, we create a new directory called `builder`; inside it, we create the `com/example/builder` directory structure. Next, we create a `module-info.java` file in the `builder` directory with the following content:

```
module com.example.builder {
    requires com.example.brickhouse;
    uses com.example.brickhouse.BrickHouse;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and uses the `BrickHouse` service.

Now, we create a `Builder.java` file inside the `com/example/builder` directory with the following content:

```
package com.example.builder;
import com.example.brickhouse.BrickHouse;
import java.util.ServiceLoader;

public class Builder {
    public static void main(String[] args) {
        ServiceLoader<BrickHouse> loader = ServiceLoader.load(BrickHouse.class);
        loader.forEach(BrickHouse::build);
    }
}
```

A Working Example

Let's consider a simple example of a modular Java application that uses services:

- `com.example.brickhouse`: A module that defines the `BrickHouse` service interface that other modules can implement
- `com.example.bricksprovider`: A module that provides an implementation of the `BrickHouse` service and declares it in its `module-info.java` file using the `provides` keyword
- `com.example.builder`: A module that consumes the `BrickHouse` service and declares the required service in its `module-info.java` file using the `uses` keyword

The builder can then use the `ServiceLoader` API to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module.

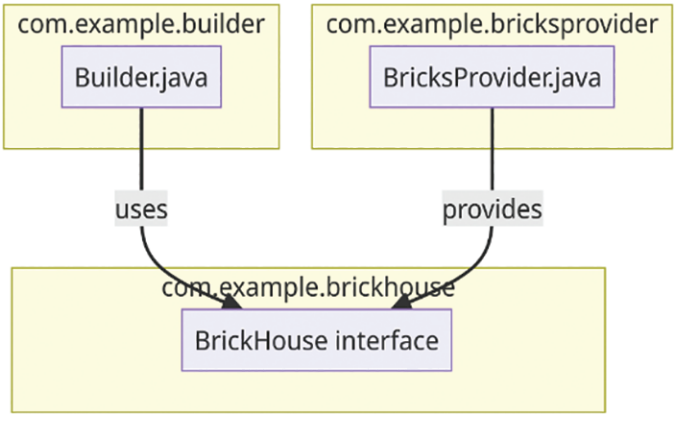

Figure 3.3 Modular Services

Figure 3.3 depicts the relationships between the modules and classes in a module diagram. The module diagram represents the dependencies and relationships between the modules and classes:

- The `com.example.builder` module contains the `Builder.java` class, which uses the `BrickHouse` interface from the `com.example.brickhouse` module.
- The `com.example.bricksprovider` module contains the `BricksProvider.java` class, which implements and provides the `BrickHouse` interface.

Implementation Details

The `ServiceLoader` API is a powerful mechanism that allows the `com.example.builder` module to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module at runtime. This allows for more flexibility and better separation of concerns between modules. The following subsections focus on some implementation details that can help us better understand the interactions between the modules and the role of the `ServiceLoader` API.

Discovering Service Implementations

The `ServiceLoader.load()` method takes a service interface as its argument—in our case, `BrickHouse.class`—and returns a `ServiceLoader` instance. This instance is an iterable object containing all available service implementations. The `ServiceLoader` relies on the information provided in the `module-info.java` files to discover the service implementations.

Instantiating Service Implementations

When iterating over the `ServiceLoader` instance, the API automatically instantiates the service implementations provided by the service providers. In our example, the `BricksProvider` class is instantiated, and its `build()` method is called when iterating over the `ServiceLoader` instance.

Encapsulating Implementation Details

By using the JPMS, the `com.example.bricksprovider` module can encapsulate its implementation details, exposing only the `BrickHouse` service that it provides. This allows the `com.example.builder` module to consume the service without depending on the concrete implementation, creating a more robust and maintainable system.

Adding More Service Providers

Our example can be easily extended by adding more service providers implementing the `BrickHouse` interface. As long as the new service providers are properly declared in their respective `module-info.java` files, the `com.example.builder` module will be able to discover and use them automatically through the `ServiceLoader` API. This allows for a more modular and extensible system that can adapt to changing requirements or new implementations.

Figure 3.4 is a use-case diagram that depicts the interactions between the service consumer and service provider. It includes two actors: Service Consumer and Service Provider.

- **Service Consumer:** This uses the services provided by the Service Provider. The Service Consumer interacts with the Modular JDK in the following ways:
  - **Discover Service Implementations:** The Service Consumer uses the Modular JDK to find available service implementations.
  - **Instantiate Service Implementations:** Once the service implementations are discovered, the Service Consumer uses the Modular JDK to create instances of these services.
  - **Encapsulate Implementation Details:** The Service Consumer benefits from the encapsulation provided by the Modular JDK, which allows it to use services without needing to know their underlying implementation.
- **Service Provider:** This implements and provides the services. The Service Provider interacts with the Modular JDK in the following ways:
  - **Implement Service Interface:** The Service Provider uses the Modular JDK to implement the service interface, which defines the contract for the service.
  - **Encapsulate Implementation Details:** The Service Provider uses the Modular JDK to hide the details of its service implementation, exposing only the service interface.
  - **Add More Service Providers:** The Service Provider can use the Modular JDK to add more providers for the service, enhancing the modularity and extensibility of the system.

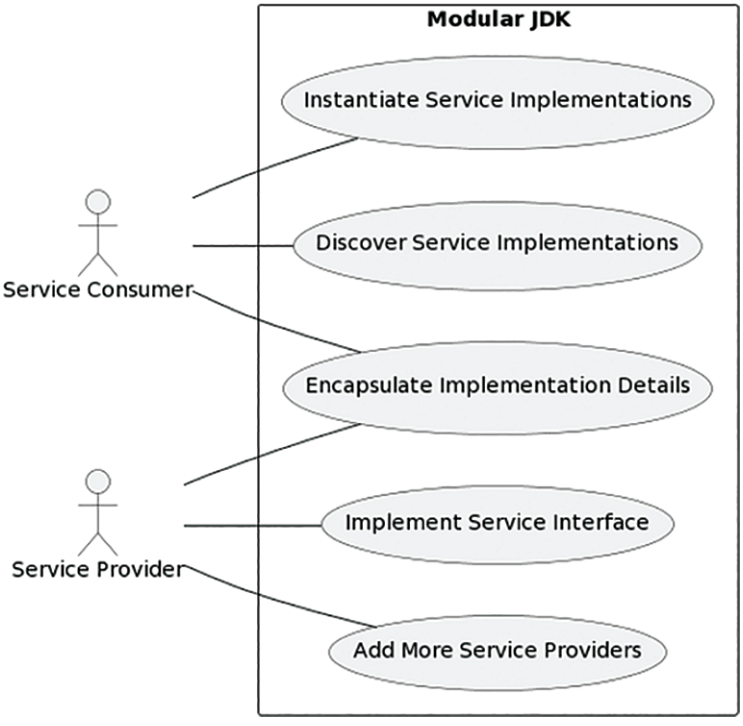

Figure 3.4 Use-Case Diagram Highlighting the Service Consumer and Service Provider

The Modular JDK acts as a robust facilitator for these interactions, establishing a comprehensive platform where service providers can effectively offer their services. Simultaneously, it provides an avenue for service consumers to discover and utilize these services efficiently. This dynamic ecosystem fosters a seamless exchange of services, enhancing the overall functionality and interoperability of modular Java applications.

JAR Hell Versioning Problem and Jigsaw Layers

Before diving into the details of the JAR hell versioning problem and Jigsaw layers, I'd like to introduce Nikita Lipski, a fellow JVM engineer and an expert in the field of Java modularity. Nikita has provided valuable insights and a comprehensive write-up on this topic, which we will be discussing in this section. His work will help us better understand the JAR hell versioning problem and how Jigsaw layers can be utilized to address this issue in JDK 11 and JDK 17.

Java's backward compatibility is one of its key features. This compatibility ensures that when a new version of Java is released, applications built for older versions can run on the new version without any changes to the source code, and often even without recompilation. The same principle applies to third-party libraries—applications can work with updated versions of the libraries without modifications to the source code.

However, this compatibility does not extend to versioning at the source level, and the JPMS does not introduce versioning at this level, either. Instead, versioning is managed at the artifact level, using artifact management systems like Maven or Gradle. These systems handle versioning and dependency management for the libraries and frameworks used in Java projects, ensuring that the correct versions of the dependencies are included in the build process. But what happens when a Java application depends on multiple third-party libraries, which in turn may depend on different versions of another library? This can lead to conflicts and runtime errors if multiple versions of the same library are present on the classpath.

So, although JPMS has certainly improved modularity and code organization in Java, the "JAR hell" problem can still be relevant when dealing with versioning at the artifact level. Let's look at an example (shown in Figure 3.5) where an application depends on two third-party libraries (Foo and Bar), which in turn depend on different versions of another library (Baz).

If both versions of the Baz library are placed on the classpath, it becomes unclear which version of the library will be used at runtime, resulting in unavoidable version conflicts. To address this issue, JPMS prohibits such situations by detecting split packages, which are not allowed in JPMS, in support of its "reliable configuration" goal (Figure 3.6).

While detecting versioning problems early is useful, JPMS does not provide a recommended way to resolve them. One approach to address these problems is to use the latest version of the conflicting library, assuming it is backward compatible. However, this might not always be possible due to introduced incompatibilities.

To address such cases, JPMS offers the *ModuleLayer* feature, which allows for the installation of a module sub-graph into the module system in an isolated manner. When different versions of the conflicting library are placed into separate layers, both of those versions can be loaded by JPMS. Although there is no direct way to access a module of the child layer from the parent layer, this can be achieved indirectly—by implementing a service provider in the child layer module, which the parent layer module can then use. (See the earlier discussion of "Implementing Modular Services with JDK 17" for more details.)

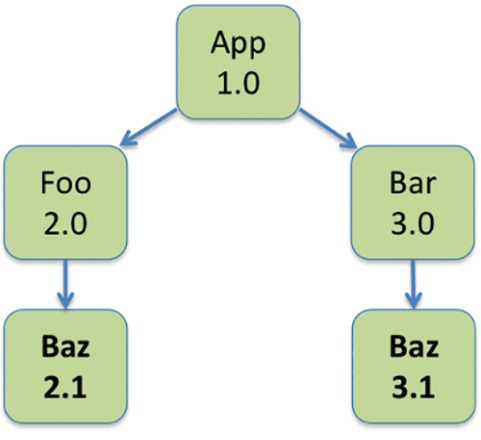

Figure 3.5 Modularity and Version Conflicts

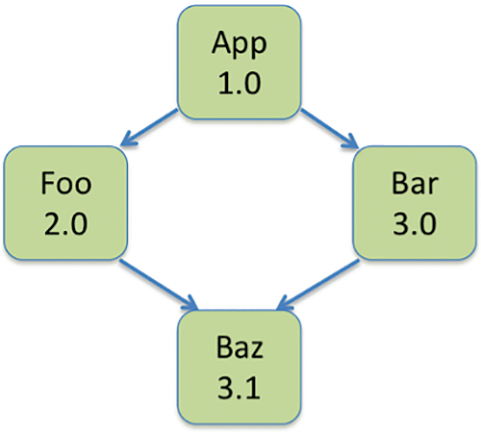

Figure 3.6 Reliable Configuration with JPMS

Working Example: JAR Hell

In this section, a working example is provided to demonstrate the use of module layers in addressing the JAR hell problem in the context of JDK 17 (this strategy is applicable to JDK 11 users as well). This example builds upon Nikita's explanation and the house service provider implementation we discussed earlier. It demonstrates how you can work with different versions of a library (termed basic and high-quality implementations) within a modular application.

First, let's take a look at the sample code provided by Java SE 9 documentation:[1]

```
1 ModuleFinder finder = ModuleFinder.of(dir1, dir2, dir3);
2 ModuleLayer parent = ModuleLayer.boot();
3 Configuration cf = parent.configuration().resolve(finder, ModuleFinder.of(),
  Set.of("myapp"));
4 ClassLoader scl = ClassLoader.getSystemClassLoader();
5 ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

In this example:

- At line 1, a `ModuleFinder` is set up to locate modules from specific directories (`dir1`, `dir2`, and `dir3`).
- At line 2, the boot layer is established as the parent layer.
- At line 3, the boot layer's configuration is resolved as the parent configuration for the modules found in the directories specified in line 1.
- At line 5, a new layer with the resolved configuration is created, using a single class loader with the system class loader as its parent.

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html

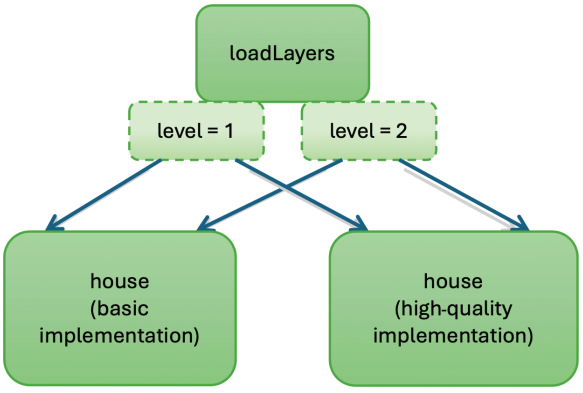

Figure 3.7 JPMS Example Versions and Layers

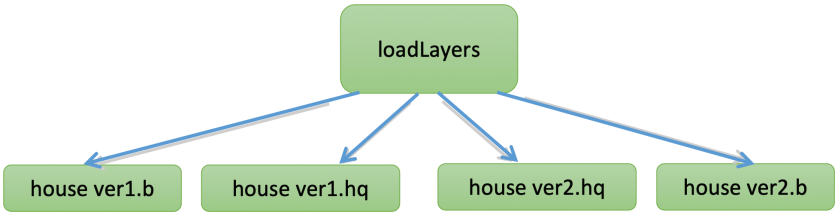

Figure 3.8 JPMS Example Version Layers Flattened

Now, let's extend our house service provider implementation. We'll have basic and high-quality implementations provided in the `com.codekaram.provider` modules. You can think of the "basic implementation" as version 1 of the house library and the "high-quality implementation" as version 2 of the house library (Figure 3.7).

For each level, we will reach out to both the libraries. So, our combinations would be level 1 + basic implementation provider, level 1 + high-quality implementation provider, level 2 + basic implementation provider, and level 2 + high-quality implementation provider. For simplicity, let's denote the combinations as *house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*, respectively (Figure 3.8).

Implementation Details

Building upon the concepts introduced by Nikita in the previous section, let's dive into the implementation details and understand how the layers' structure and program flow work in practice. First, let's look at the source trees:

```
ModuleLayer
├── basic
│   └── src
│       └── com.codekaram.provider
```

```
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
├── high-quality
│   └── src
│       └── com.codekaram.provider
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
└── src
    └── com.codekaram.brickhouse
        ├── classes
        │   ├── com
        │   │   └── codekaram
        │   │       └── brickhouse
        │   │           ├── loadLayers.java
        │   │           └── spi
        │   │               └── BricksProvider.java
        │   └── module-info.java
        └── tests
```

Here's the module file information and the module graph for `com.codekaram.provider`. Note that these look exactly the same for both the basic and high-quality implementations.

```
module com.codekaram.provider {
    requires com.codekaram.brickhouse;
    uses com.codekaram.brickhouse.spi.BricksProvider;
    provides com.codekaram.brickhouse.spi.BricksProvider with com.codekaram.provider.House;
}
```

The module diagram (shown in Figure 3.9) helps visualize the dependencies between modules and the services they provide, which can be useful for understanding the structure of a modular Java application:

- The `com.codekaram.provider` module depends on the `com.codekaram.brickhouse` module and implicitly depends on the `java.base` module, which is the foundational module of every Java application. This is indicated by the arrows pointing from `com`.

`codekaram.provider` to `com.codekaram.brickhouse` and the assumed arrow to `java.base`.

- The `com.codekaram.brickhouse` module also implicitly depends on the `java.base` module, as all Java modules do.

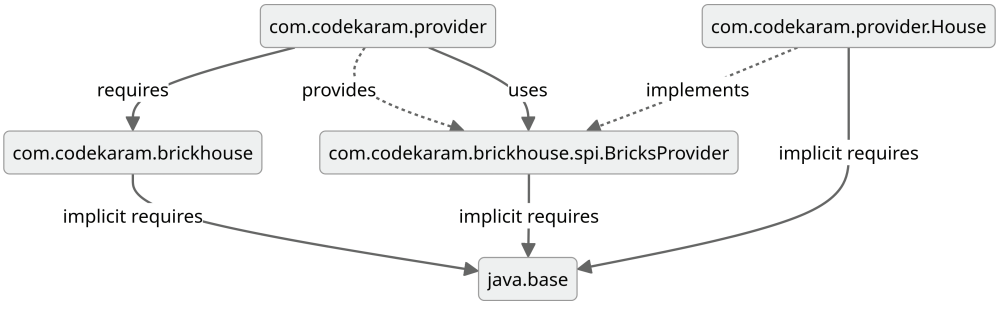

Figure 3.9 A Working Example with Services and Layers

- The `java.base` module does not depend on any other module and is the core module upon which all other modules rely.
- The `com.codekaram.provider` module provides the service `com.codekaram.brickhouse.spi.BricksProvider` with the implementation `com.codekaram.provider.House`. This relationship is represented in the graph by a dashed arrow from `com.codekaram.provider` to `com.codekaram.brickhouse.spi.BricksProvider`.

Before diving into the code for these providers, let's look at the module file information for the `com.codekaram.brickhouse` module:

```
module com.codekaram.brickhouse {
    uses com.codekaram.brickhouse.spi.BricksProvider;
    exports com.codekaram.brickhouse.spi;
}
```

The `loadLayers` class will not only handle forming layers, but also be able to load the service providers for each level. That's a bit of a simplification, but it helps us to better understand the flow. Now, let's examine the `loadLayers` implementation. Here's the creation of the layers' code based on the sample code from the "Working Example: JAR Hell" section:

```
static ModuleLayer getProviderLayer(String getCustomDir) {
   ModuleFinder finder = ModuleFinder.of(Paths.get(getCustomDir));
   ModuleLayer parent = ModuleLayer.boot();
   Configuration cf = parent.configuration().resolve(finder,
     ModuleFinder.of(), Set.of("com.codekaram.provider"));
   ClassLoader scl = ClassLoader.getSystemClassLoader();
   ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

```
   System.out.println("Created a new layer for " + layer);
   return layer;
}
```

If we simply want to create two layers, one for house version `basic` and another for house version `high-quality`, all we have to do is call `getProviderLayer()` (from the `main` method):

```
doWork(Stream.of(args)
       .map(getCustomDir -> getProviderLayer(getCustomDir)));
```

If we pass the two directories `basic` and `high-quality` as runtime parameters, the `getProviderLayer()` method will look for `com.codekaram.provider` in those directories and then create a layer for each. Let's examine the output (the line numbers have been added for the purpose of clarity and explanation):

```
1  $ java --module-path mods -m com.codekaram.brickhouse/
   com.codekaram.brickhouse.loadLayers basic high-quality

2  Created a new layer for com.codekaram.provider

3  I am the basic provider

4  Created a new layer for com.codekaram.provider

5  I am the high-quality provider
```

- Line 1 is our command-line argument with `basic` and `high-quality` as directories that provide the implementation of the `BrickProvider` service.
- Lines 2 and 4 are outputs indicating that `com.codekaram.provider` was found in both the directories and a new layer was created for each.
- Lines 3 and 5 are the output of `provider.getName()` as implemented in the `doWork()` code:

```
private static void doWork(Stream<ModuleLayer> myLayers){
myLayers.flatMap(moduleLayer -> ServiceLoader
   .load(moduleLayer, BricksProvider.class)
    .stream().map(ServiceLoader.Provider::get))
  .forEach(eachSLProvider -> System.out.println("I am the " + eachSLProvider.getName() +
" provider"));}
```

In `doWork()`, we first create a service loader for the `BricksProvider` service and load the provider from the module layer. We then print the return `String` of the `getName()` method for that provider. As seen in the output, we have two module layers and we were successful in printing the `I am the basic provider` and `I am the high-quality provider` outputs, where `basic` and `high-quality` are the return strings of the `getName()` method.

Now, let's visualize the workings of the four layers that we discussed earlier. To do so, we'll create a simple problem statement that builds a quote for basic and high-quality bricks for both levels of the house. First, we add the following code to our `main()` method:

```
int[] level = {1,2};
IntStream levels = Arrays.stream(level);
```

Next, we stream `doWork()` as follows:

```
levels.forEach(levelcount -> loadLayers
         .doWork(…
```

We now have four layers similar to those mentioned earlier (*house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*). Here's the updated output:

```
Created a new layer for com.codekaram.provider
My basic 1 level house will need 18000 bricks and those will cost me $6120
Created a new layer for com.codekaram.provider
My high-quality 1 level house will need 18000 bricks and those will cost me $9000
Created a new layer for com.codekaram.provider
My basic 2 level house will need 36000 bricks and those will cost me $12240
Created a new layer for com.codekaram.provider
My high-quality 2 level house will need 36000 bricks and those will be over my budget of $15000
```

NOTE The return string of the `getName()` methods for our providers has been changed to return just the `"basic"` and `"high-quality"` strings instead of an entire sentence.

The variation in the last line of the updated output serves as a demonstration of how additional conditions can be applied to service providers. Here, a budget constraint check has been integrated into the high-quality provider's implementation for a two-level house. You can, of course, customize the output and conditions as per your requirements.

Here's the updated `doWork()` method to handle both the level and the provider, along with the relevant code in the main method:

```
private static void doWork(int level, Stream<ModuleLayer> myLayers){
    myLayers.flatMap(moduleLayer -> ServiceLoader
        .load(moduleLayer, BricksProvider.class)
        .stream().map(ServiceLoader.Provider::get))
    .forEach(eachSLProvider -> System.out.println("My " + eachSLProvider.getName()
    + " " + level + " level house will need " + eachSLProvider.getBricksQuote(level)));
}
```

```
public static void main(String[] args) {
    int[] levels = {1, 2};
    IntStream levelStream = Arrays.stream(levels);

    levelStream.forEach(levelcount -> doWork(levelcount, Stream.of(args)
        .map(getCustomDir -> getProviderLayer(getCustomDir))));
}
```

Now, we can calculate the number of bricks and their cost for different levels of the house using the `basic` and `high-quality` implementations, with a separate module layer being devoted to each implementation. This demonstrates the power and flexibility that module layers provide, by enabling you to dynamically load and unload different implementations of a service without affecting other parts of your application.

Remember to adjust the service providers' code based on your specific use case and requirements. The example provided here is just a starting point for you to build on and adapt as needed.

In summary, this example illustrates the utility of Java module layers in creating applications that are both adaptable and scalable. By using the concepts of module layers and the Java `ServiceLoader`, you can create extensible applications, allowing you to adapt those applications to different requirements and conditions without affecting the rest of your codebase.

Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi) has been an alternative module system available to Java developers since 2000, long before the introduction of Jigsaw and Java module layers. As there was no built-in standard module system in Java at the time of OSGi's emergence, it addressed many modularity problems differently compared to Project Jigsaw. In this section, with insights from Nikita, whose expertise in Java modularity encompasses OSGi, we will compare Java module layers and OSGi, highlighting their similarities and differences.

OSGi Overview

OSGi is a mature and widely used framework that provides modularity and extensibility for Java applications. It offers a dynamic component model, which allows developers to create, update, and remove modules (called bundles) at runtime without restarting the application.

Similarities

- **Modularity:** Both Java module layers and OSGi promote modularity by enforcing a clear separation between components, making it easier to maintain, extend, and reuse code.
- **Dynamic loading:** Both technologies support dynamic loading and unloading of modules or bundles, allowing developers to update, extend, or remove components at runtime without affecting the rest of the application.

- **Service abstraction:** Both Java module layers (with the `ServiceLoader`) and OSGi provide service abstractions that enable loose coupling between components. This allows for greater flexibility when switching between different implementations of a service.

Differences

- **Maturity:** OSGi is a more mature and battle-tested technology, with a rich ecosystem and tooling support. Java module layers, which were introduced in JDK 9, are comparatively newer and may not have the same level of tooling and library support as OSGi.
- **Integration with Java platform:** Java module layers are a part of the Java platform, providing a native solution for modularity and extensibility. OSGi, by contrast, is a separate framework that builds on top of the Java platform.
- **Complexity:** OSGi can be more complex than Java module layers, with a steeper learning curve and more advanced features. Java module layers, while still providing powerful functionality, may be more straightforward and easier to use for developers who are new to modularity concepts.
- **Runtime environment:** OSGi applications run inside an OSGi container, which manages the life cycle of the bundles and enforces modularity rules. Java module layers run directly on the Java platform, with the module system handling the loading and unloading of modules.
- **Versioning:** OSGi provides built-in support for multiple versions of a module or bundle, allowing developers to deploy and run different versions of the same component concurrently. This is achieved by qualifying modules with versions and applying "uses constraints" to ensure safe class namespaces exist for each module. However, dealing with versions in OSGi can introduce unnecessary complexity for module resolution and for end users. In contrast, Java module layers do not natively support multiple versions of a module, but you can achieve similar functionality by creating separate module layers for each version.
- **Strong encapsulation:** Java module layers, as first-class citizens in the JDK, provide strong encapsulation by issuing error messages when unauthorized access to non-exported functionality occurs, even via reflection. In OSGi, non-exported functionality can be "hidden" using class loaders, but the module internals are still available for reflection access unless a special security manager is set. OSGi was limited by pre-JPMS features of Java SE and could not provide the same level of strong encapsulation as Java module layers.

When it comes to achieving modularity and extensibility in Java applications, developers typically have two main options: Java module layers and OSGi. Remember, the choice between Java module layers and OSGi is not always binary and can depend on many factors. These include the specific requirements of your project, the existing technology stack, and your team's familiarity with the technologies. Also, it's worth noting that Java module layers and OSGi are not the only options for achieving modularity in Java applications. Depending on your specific needs and context, other solutions might be more appropriate. It is crucial to thoroughly evaluate the pros and cons of all available options before making a decision for your project. Your choice should be based on the specific demands and restrictions of your project to ensure optimal outcomes.

On the one hand, if you need advanced features like multiple version support and a dynamic component model, OSGi may be the better option for you. This technology is ideal for complex applications that require both flexibility and scalability. However, it can be more difficult to learn and implement than Java module layers, so it may not be the best choice for developers who are new to modularity.

On the other hand, Java module layers offer a more straightforward solution for achieving modularity and extensibility in your Java applications. This technology is built into the Java platform itself, which means that developers who are already familiar with Java should find it relatively easy to use. Additionally, Java module layers offer strong encapsulation features that can help prevent dependencies from bleeding across different modules.

Introduction to Jdeps, Jlink, Jdeprscan, and Jmod

This section covers four tools that aid in the development and deployment of modular applications: *jdeps*, *jlink*, *jdeprscan*, and *jmod*. Each of these tools serves a unique purpose in the process of building, analyzing, and deploying Java applications.

Jdeps

Jdeps is a tool that facilitates analysis of the dependencies of Java classes or packages. It's particularly useful when you're trying to create a module file for JAR files. With *jdeps*, you can create various filters using regular expressions (*regex*); a regular expression is a sequence of characters that forms a search pattern. Here's how you can use *jdeps* on the `loadLayers` class:

```
$ jdeps mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse -> com.codekaram.brickhouse.spi   not found
   com.codekaram.brickhouse -> java.io                        java.base
   com.codekaram.brickhouse -> java.lang                      java.base
   com.codekaram.brickhouse -> java.lang.invoke               java.base
   com.codekaram.brickhouse -> java.lang.module               java.base
   com.codekaram.brickhouse -> java.nio.file                  java.base
   com.codekaram.brickhouse -> java.util                      java.base
   com.codekaram.brickhouse -> java.util.function             java.base
   com.codekaram.brickhouse -> java.util.stream               java.base
```

The preceding command has the same effect as passing the option `-verbose:package` to *jdeps*. The `-verbose` option by itself will list all the dependencies:

```
$ jdeps -v mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse.loadLayers -> com.codekaram.brickhouse.spi.BricksProvider  not found
   com.codekaram.brickhouse.loadLayers -> java.io.PrintStream                  java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.Class                      java.base
```

```
com.codekaram.brickhouse.loadLayers -> java.lang.ClassLoader                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.ModuleLayer                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.NoSuchMethodException      java.base
com.codekaram.brickhouse.loadLayers -> java.lang.Object                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.String                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.System                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.CallSite            java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.LambdaMetafactory   java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandle        java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles$Lookup java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodType          java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.StringConcatFactory java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.Configuration       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.ModuleFinder        java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Path                   java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Paths                  java.base
com.codekaram.brickhouse.loadLayers -> java.util.Arrays                     java.base
com.codekaram.brickhouse.loadLayers -> java.util.Collection                 java.base
com.codekaram.brickhouse.loadLayers -> java.util.ServiceLoader              java.base
com.codekaram.brickhouse.loadLayers -> java.util.Set                        java.base
com.codekaram.brickhouse.loadLayers -> java.util.Spliterator                java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Consumer          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Function          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.IntConsumer       java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Predicate         java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.IntStream           java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.Stream              java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.StreamSupport       java.base
```

Jdeprscan

Jdeprscan is a tool that analyzes the usage of deprecated APIs in modules. Deprecated APIs are older APIs that the Java community has replaced with newer ones. These older APIs are still supported but are marked for removal in future releases. *Jdeprscan* helps developers maintain their code by suggesting alternative solutions to these deprecated APIs, aiding them in transitioning to newer, supported APIs.

Here's how you can use *jdeprscan* on the `com.codekaram.brickhouse` module:

```
$ jdeprscan --for-removal mods/com.codekaram.brickhouse
No deprecated API marked for removal found.
```

In this example, *jdeprscan* is used to scan the `com.codekaram.brickhouse` module for deprecated APIs that are marked for removal. The output indicates that no such deprecated APIs are found.

You can also use `--list` to see all deprecated APIs in a module:

```
$ jdeprscan --list mods/com.codekaram.brickhouse
No deprecated API found.
```

In this case, no deprecated APIs are found in the `com.codekaram.brickhouse` module.

Jmod

Jmod is a tool used to create, describe, and list JMOD files. JMOD files are an alternative to JAR files for packaging modular Java applications, which offer additional features such as native code and configuration files. These files can be used for distribution or to create custom runtime images with *jlink*.

Here's how you can use *jmod* to create a JMOD file for the `brickhouse` example. Let's first compile and package the module specific to this example:

```
$ javac --module-source-path src -d build/modules $(find src -name "*.java")
$ jmod create --class-path build/modules/com.codekaram.brickhouse com.codekaram.brickhouse.jmod
```

Here, the `jmod create` command is used to create a JMOD file named `com.codekaram.brickhouse.jmod` from the `com.codekaram.brickhouse` module located in the `build/modules` directory. You can then use the `jmod describe` command to display information about the JMOD file:

```
$ jmod describe com.codekaram.brickhouse.jmod
```

This command will output the module descriptor and any additional information about the JMOD file.

Additionally, you can use the `jmod list` command to display the contents of the created JMOD file:

```
$ jmod list com.codekaram.brickhouse.jmod
com/codekaram/brickhouse/
com/codekaram/brickhouse/loadLayers.class
com/codekaram/brickhouse/loadLayers$1.class
…
```

The output lists the contents of the `com.codekaram.brickhouse.jmod` file, showing the package structure and class files.

By using *jmod* to create JMOD files, describe their contents, and list their individual files, you can gain a better understanding of your modular application's structure and streamline the process of creating custom runtime images with *jlink*.

Jlink

Jlink is a tool that helps link modules and their transitive dependencies to create custom modular runtime images. These custom images can be packaged and deployed without needing the entire Java Runtime Environment (JRE), which makes your application lighter and faster to start.

To use the *jlink* command, this tool needs to be added to your path. First, ensure that the `$JAVA_HOME/bin` is in the path. Next, type `jlink` on the command line:

```
$ jlink
Error: --module-path must be specified
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Use --help for a list of possible options
```

Here's how you can use *jlink* for the code shown in "Implementing Modular Services with JDK 17":

```
$ jlink --module-path $JAVA_HOME/jmods:build/modules --add-modules com.example.builder --output
consumer.services --bind-services
```

A few notes on this example:

- The command includes a directory called `$JAVA_HOME/jmods` in the `module-path`. This directory contains the `java.base.jmod` needed for all application modules.
- Because the module is a consumer of services, it's necessary to link the service providers (and their dependencies). Hence, the `–bind-services` option is used.
- The runtime image will be available in the `consumer.services` directory as shown here:

```
$ ls consumer.services/
bin conf      include   legal     lib       release
```

Let's now run the image:

```
$ consumer.services/bin/java -m com.example.builder/com.example.builder.Builder
Building a house with bricks...
```

With *jlink*, you can create lightweight, custom, stand-alone runtime images tailored to your modular Java applications, thereby simplifying the deployment and reducing the size of your distributed application.

Conclusion

This chapter has undertaken a comprehensive exploration of Java modules, tools, and techniques to create and manage modular applications. We have delved into the Java Platform Module System (JPMS), highlighting its benefits such as reliable configuration and strong encapsulation. These features contribute to more maintainable and scalable applications.

We navigated the intricacies of creating, packaging, and managing modules, and explored the use of module layers to enhance application flexibility. These practices can help address common challenges faced when migrating to newer JDK versions (e.g., JDK 11 or JDK 17), including updating project structures and ensuring dependency compatibility.

Performance Implications

The use of modular Java carries significant performance implications. By including only the necessary modules in your application, the JVM loads fewer classes, which improves start-up performance and reduces the memory footprint. This is particularly beneficial in resource-limited environments such as microservices running in containers. However, it is important to note that while modularity can improve performance, it also introduces a level of complexity. For instance, improper module design can lead to cyclic dependencies,[2] negatively impacting performance. Therefore, careful design and understanding of modules are essential to fully reap the performance benefits.

Tools and Future Developments

We examined the use of powerful tools like *jdeps*, *jdeprscan*, *jmod*, and *jlink*, which are instrumental in identifying and addressing compatibility issues, creating custom runtime images, and streamlining the deployment of modular applications. Looking ahead, we can anticipate more advanced options for creating custom runtime images with *jlink*, and more detailed and accurate dependency analysis with *jdeps*.

As more developers adopt modular Java, new best practices and patterns will emerge, alongside new tools and libraries designed to work with JPMS. The Java community is continuously improving JPMS, with future Java versions expected to refine and expand its capabilities.

Embracing the Modular Programming Paradigm

Transitioning to modular Java can present unique challenges, especially in understanding and implementing modular structures in large-scale applications. Compatibility issues may arise with third-party libraries or frameworks that may not be fully compatible with JPMS. These challenges, while part of the journey toward modernization, are often outweighed by the benefits of modular Java, such as improved performance, enhanced scalability, and better maintainability.

In conclusion, by leveraging the knowledge gained from this chapter, you can confidently migrate your projects and fully harness the potential of modular Java applications. The future of modular Java is exciting, and embracing this paradigm will equip you to meet the evolving needs of the software development landscape. It's an exciting time to be working with modular Java, and we look forward to seeing how it evolves and shapes the future of robust and efficient Java applications.

[2] https://openjdk.org/projects/jigsaw/spec/issues/#CyclicDependences

This page intentionally left blank

Chapter 4

The Unified Java Virtual Machine Logging Interface

The Java HotSpot Virtual Machine (VM) is a treasure trove of features and flexibility. Command-line options allow for enabling its experimental or diagnostic features and exploring various VM and garbage collection activities. Within the HotSpot VM, there are distinct builds tailored for different purposes. The primary version is the *product* build, optimized for performance and used in production environments. In addition, there are *debug* builds, which include the *fast-debug* and the *slow-debug* variants.

> **NOTE** The asserts-enabled *fast-debug* build offers additional debugging capabilities with minimal performance overhead, making it suitable for development environments where some level of diagnostics is needed without significantly compromising performance. On the other hand, the *slow-debug* build includes comprehensive debugging tools and checks, ideal for in-depth analysis with native debuggers but with a significant performance trade-off.

Until JDK 9, the logging interface of the HotSpot VM was fragmented, lacking unification. Developers had to combine various print options to retrieve detailed logging information with appropriate timestamps and context. This chapter guides you through the evolution of this feature, focusing on the unified logging system introduced in JDK 9, which was further enhanced in JDK 11 and JDK 17.

The Need for Unified Logging

The Java HotSpot VM is rich in options, but its lack of a unified logging interface was a significant shortcoming. Various command-line options were necessary to retrieve specific logging details, which could have been clearer and more efficient. A brief overview of these options illustrates the complexity (many more options were available, aside from those listed here):

- Garbage collector (GC) event timestamp: `-XX:+PrintGCTimeStamps` or `-XX:+PrintGCDateStamps`
- GC event details: `-XX:+PrintGCDetails`
- GC event cause: `-XX:+PrintGCCause`

- GC adaptiveness: `-XX:+PrintAdaptiveSizePolicy`
- GC-specific options (e.g., for the Garbage First [G1] GC): `-XX:+G1PrintHeapRegions`
- Compilation levels: `-XX:+PrintCompilation`
- Inlining information: `-XX:+PrintInlining`
- Assembly code: `-XX:+PrintAssembly` (requires `-XX:+UnlockDiagnosticVMOptions`)
- Class histogram: `-XX:+PrintClassHistogram`
- Information on `java.util.concurrent` locks: `-XX:+PrintConcurrentLocks`

Memorizing each option and determining whether the information is at the info, debug, or tracing level is challenging. Thus, JDK Enhancement Proposal (JEP) 158: *Unified JVM Logging*[1] was introduced in JDK 9. It was followed by JEP 271: *Unified GC Logging*,[2] which offered a comprehensive logging framework for GC activities.

Unification and Infrastructure

The Java HotSpot VM has a unified logging interface that provides a familiar look and approach to all logging information for the JVM. All log messages are characterized using tags. This enables you to request the level of information necessary for your level of investigation, from "error" and "warning" logs to "info," "trace," and "debug" levels.

To enable JVM logging, you need to provide the `-Xlog` option. By default (i.e., when no `-Xlog` is specified), only warnings and errors are logged to *stderr*. So, the default configuration looks like this:

```
-Xlog:all=warning:stderr:uptime,level,tags
```

Here, we see `all`, which is simply an alias for all tags.

The unified logging system is an all-encompassing system that provides a consistent interface for all JVM logging. It is built around four main components:

- **Tags:** Used to categorize log messages, with each message assigned one or more tags. Tags make it easier to filter and find specific log messages. Some of the predefined tags include `gc`, `compiler`, and `threads`.
- **Levels:** Define the severity or importance of a log message. Six levels are available: `debug`, `error`, `info`, `off`, `trace`, and `warning`.
- **Decorators:** Provide additional context to a log message, such as timestamps, process IDs, and more.
- **Outputs:** The destinations where the log messages are written, such as *stdout*, *stderr*, or a file.

[1] https://openjdk.org/jeps/158

[2] https://openjdk.org/jeps/271

Performance Metrics

In the context of JVM performance engineering, the unified logging system provides valuable insights into various performance metrics. These metrics include GC activities, just-in-time (JIT) compilation events, and system resource usage, among others. By monitoring these metrics, developers can identify potential performance issues and preempt necessary actions to optimize the performance of their Java applications.

Tags in the Unified Logging System

In the unified logging system, tags are crucial for segregating and identifying different types of log messages. Each tag corresponds to a particular system area or a specific type of operation. By enabling specific tags, users can tailor the logging output to focus on areas of interest.

Log Tags

The unified logging system provides a myriad of tags to capture granular information. Tags such as `gc` for garbage collection, `thread` for thread operations, `class` for class loading, `cpu` for CPU-related events, `os` for operating system interactions, and many more as highlighted here:

```
add, age, alloc, annotation, arguments, attach, barrier, biasedlocking, blocks, bot, breakpoint,
bytecode, cds, census, class, classhisto, cleanup, codecache, compaction, compilation, condy,
constantpool, constraints, container, coops, cpu, cset, data, datacreation, dcmd, decoder,
defaultmethods, director, dump, dynamic, ergo, event, exceptions, exit, fingerprint, free,
freelist, gc, handshake, hashtables, heap, humongous, ihop, iklass, indy, init, inlining, install,
interpreter, itables, jfr, jit, jni, jvmci, jvmti, lambda, library, liveness, load, loader,
logging, malloc, map, mark, marking, membername, memops, metadata, metaspace, methodcomparator,
methodhandles, mirror, mmu, module, monitorinflation, monitormismatch, nestmates, nmethod,
nmt, normalize, numa, objecttagging, obsolete, oldobject, oom, oopmap, oops, oopstorage, os,
owner, pagesize, parser, patch, path, perf, periodic, phases, plab, placeholders, preorder,
preview, promotion, protectiondomain, ptrqueue, purge, record, redefine, ref, refine, region,
reloc, remset, resolve, safepoint, sampling, scavenge, sealed, setting, smr, stackbarrier,
stackmap, stacktrace, stackwalk, start, startup, startuptime, state, stats, streaming,
stringdedup, stringtable, subclass, survivor, suspend, sweep, symboltable, system, table, task,
thread, throttle, time, timer, tlab, tracking, unload, unshareable, update, valuebasedclasses,
verification, verify, vmmutex, vmoperation, vmthread, vtables, vtablestubs, workgang
```

Specifying `all` instead of a tag combination matches all tag combinations.

If you want to run your application with all log messages, you could execute your program with `-Xlog`, which is equivalent to `-Xlog:all`. This option will log all messages to *stdout* with the level set at `info`, while warnings and errors will go to *stderr*.

Specific Tags

By default, running your application with `-Xlog` but without specific tags will produce a large volume of log messages from various system parts. If you want to focus on specific areas, you can do so by specifying the tags.

For instance, if you are interested in log messages related to garbage collection, the heap, and compressed oops, you could specify the `gc` tag. Running your application in JDK 17 with the `-Xlog:gc*` option would produce log messages related to garbage collection:

```
$ java -Xlog:gc* LockLoops
[0.006s][info][gc] Using G1
[0.007s][info][gc,init] Version: 17.0.8+9-LTS-211 (release)
[0.007s][info][gc,init] CPUs: 16 total, 16 available
…
[0.057s][info][gc,metaspace] Compressed class space mapped at: 0x0000000132000000
-0x0000000172000000, reserved size: 1073741824
[0.057s][info][gc,metaspace] Narrow klass base: 0x0000000131000000, Narrow klass shift: 0, Narrow
klass range: 0x100000000
```

Similarly, you might be interested in the `thread` tag if your application involves heavy multi-threading. Running your application with the `-Xlog:thread*` option would produce log messages related to thread operations:

```
$ java -Xlog:thread* LockLoops
[0.019s][info][os,thread] Thread attached (tid: 7427, pthread id: 123145528827904).
[0.030s][info][os,thread] Thread "GC Thread#0" started (pthread id: 123145529888768, attributes:
stacksize: 1024k, guardsize: 4k, detached).
…
```

Identifying Missing Information

In some situations, enabling a tag doesn't produce the expected log messages. For instance, running your application with `-Xlog:monitorinflation*` might produce scant output, as shown here:

```
…
[11.301s][info][monitorinflation] deflated 1 monitors in 0.0000023 secs
[11.301s][info][monitorinflation] end deflating: in_use_list stats: ceiling=11264, count=2, max=3
…
```

In such cases, you may need to adjust the log level—a topic that we'll explore in the next section.

Diving into Levels, Outputs, and Decorators

Let's take a closer look at the available log levels, examine the use of decorators, and discuss ways to redirect outputs.

Levels

The JVM's unified logging system provides various logging levels, each corresponding to a different amount of detail. To understand these levels, you can use the `-Xlog:help` command, which offers information on all available options:

```
Available log levels:
 off, trace, debug, info, warning, error
```

Here,

- **`off`:** Disable logging.
- **`error`:** Critical issues within the JVM.
- **`warning`:** Potential issues that might warrant attention.
- **`info`:** General information about JVM operations.
- **`debug`:** Detailed information useful for debugging. This level provides in-depth insights into the JVM's behavior and is typically used when troubleshooting specific issues.
- **`trace`:** The most verbose level, providing extremely detailed logging. Trace level is often used for the most granular insights into JVM operations, especially when a detailed step-by-step account of events is required.

As shown in our previous examples, the default log level is set to `info` if no specific log level is given for a tag. When we ran our test, we observed very little information for the `monitorinflation` tag when we used the default `info` log level. We can explicitly set the log level to `trace` to access additional information, as shown here:

```
$ java -Xlog:monitorinflation*=trace LockLoops
```

The output logs now contain detailed information about the various locks, as shown in the following log excerpt:

```
…
[3.073s][trace][monitorinflation      ] Checking in_use_list:
[3.073s][trace][monitorinflation      ] count=3, max=3
[3.073s][trace][monitorinflation      ] in_use_count=3 equals ck_in_use_count=3
[3.073s][trace][monitorinflation      ] in_use_max=3 equals ck_in_use_max=3
[3.073s][trace][monitorinflation      ] No errors found in in_use_list checks.
[3.073s][trace][monitorinflation      ] In-use monitor info:
[3.073s][trace][monitorinflation      ] (B -> is_busy, H -> has hash code, L -> lock status)
```

```
[3.073s][trace][monitorinflation      ]          monitor  BHL            object       object type
[3.073s][trace][monitorinflation      ] ================  ===  ================  =================
[3.073s][trace][monitorinflation      ] 0x00006000020ec680  100  0x000000070fe190e0  java.
lang.ref.ReferenceQueue$Lock (is_busy: waiters=1, contentions=0owner=0x0000000000000000,
cxq=0x0000000000000000, EntryList=0x0000000000000000)
…
```

In the unified logging system, log levels are hierarchical. Hence, setting the log level to `trace` will include messages from all lower levels (`debug`, `info`, `warning`, `error`) as well. This ensures a comprehensive logging output, capturing a wide range of details, as can be seen further down in the log:

```
…
[11.046s][trace][monitorinflation      ] deflate_monitor: object=0x000000070fe70a58,
mark=0x00006000020e00d2, type='java.lang.Object'
[11.046s][debug][monitorinflation      ] deflated 1 monitors in 0.0000157 secs
[11.046s][debug][monitorinflation      ] end deflating: in_use_list stats: ceiling=11264, count=2,
max=3
[11.046s][debug][monitorinflation      ] begin deflating: in_use_list stats: ceiling=11264,
count=2, max=3
[11.046s][debug][monitorinflation      ] deflated 0 monitors in 0.0000004 secs
…
```

This hierarchical logging is a crucial aspect of the JVM's logging system, allowing for flexible and detailed monitoring based on your requirements.

Decorators

Decorators are another critical aspect of JVM logging. Decorators enrich log messages with additional context, making them more informative. Specifying `-Xlog:help` on the command line will yield the following decorators:

```
…
Available log decorators:
 time (t), utctime (utc), uptime (u), timemillis (tm), uptimemillis (um), timenanos (tn),
uptimenanos (un), hostname (hn), pid (p), tid (ti), level (l), tags (tg)
 Decorators can also be specified as 'none' for no decoration.
…
```

These decorators provide specific information in the logs. By default, `uptime`, `level`, and `tags` are selected, which is why we saw these three decorators in the log output in our earlier examples. However, you might sometimes want to use different decorators, such as

- **`pid` (`p`):** Process ID. Useful in distinguishing logs from different JVM instances.

- **`tid` (`ti`):** Thread ID. Crucial for tracing actions of specific threads, aiding in debugging concurrency issues or thread-specific behavior.
- **`uptimemillis` (`um`):** JVM uptime in milliseconds. Helps in correlating events with the timeline of the application's operation, especially in performance monitoring.

Here's how we can add the decorators:

```
$ java -Xlog:gc*::uptimemillis,pid,tid LockLoops
[0.006s][32262][8707] Using G1
[0.008s][32262][8707] Version: 17.0.8+9-LTS-211 (release)
[0.008s][32262][8707] CPUs: 16 total, 16 available [0.023s][32033][9987] Memory: 16384M
…
[0.057s][32262][8707] Compressed class space mapped at: 0x000000012a000000-0x000000016a000000,
reserved size: 1073741824
[0.057s][32262][8707] Narrow klass base: 0x0000000129000000, Narrow klass shift: 0, Narrow klass
range: 0x100000000
```

Decorators like `pid` and `tid` are particularly valuable in debugging. When multiple JVMs are running, `pid` helps identify which JVM instance generated the log. `tid` is crucial in multi-threaded applications for tracking the behavior of individual threads, which is vital in diagnosing issues like deadlocks or race conditions.

Use `uptime` or `uptimemillis` for a time reference for events, enabling a clear timeline of operations. `uptimemillis` is essential in performance analysis to understand when specific events occur relative to the application's start time. Similarly, `hostname` can be valuable in distributed systems to identify the source machine of log entries.

Decorators enhance log readability and analysis in several ways:

- **Contextual clarity:** By providing additional details such as timestamps, process IDs, and thread IDs, decorators make logs self-contained and more understandable.
- **Filtering and searching:** With relevant decorators, log data can be filtered more effectively, enabling quicker isolation of issues.
- **Correlation and causation analysis:** Decorators allow for correlating events across different parts of the system, aiding in root-cause analysis and system-wide performance optimization.

In summary, decorators in the unified logging system play a pivotal role in enhancing the debugging and performance monitoring capabilities of JVM logs. They add necessary context and clarity, making log data more actionable and insightful for developers.

Outputs

In JVM logging, outputs determine where your log information is directed. The JVM provides flexibility in controlling the output of the logs, allowing users to redirect the output to *stdout*, *stderr*, or a specific file. The default behavior is to route all warnings and errors to *stderr*, while

all `info`, `debug`, and `trace` levels are directed to *stdout*. However, users can adjust this behavior to suit their needs. For instance, a user can specify a particular file to write the log data, which can be useful when the logs need to be analyzed later or when *stdout*/*stderr* is not convenient or desired for log data.

A file name can be passed as a parameter to the `-Xlog` command to redirect the log output to that file. For example, suppose you want to log the garbage collection (`gc`) activities at the `info` level and want these logs to be written to the `gclog.txt` file. Here's an example for the `gc*` tag and level set to `info`:

```
$ java -Xlog:gc*=info:file=gclog.txt LockLoops
```

In this example, all log information about the garbage collection process at the `info` level will be written to the `gclog.txt` file. The same can be done for any tag or log level, providing tremendous flexibility to system administrators and developers.

You can even direct different log levels to different outputs, as in the following example:

```
$ java -Xlog:monitorinflation*=trace:file=lockinflation.txt -Xlog:gc*=info:file=gclog.txt
LockLoops
```

In this case, the `info`-level logs of the garbage collection process are written into `gclog.txt`, and the `trace`-level logs for `monitorinflation` are directed to `lockinflation.txt`. This allows a fine-grained approach to managing log data and enables the segregation of log data based on its verbosity level, which can be extremely helpful during troubleshooting.

> **NOTE** Directing logs to a file rather than *stdout*/*stderr* can have performance implications, particularly for I/O-intensive applications. Writing to a file may be slower and could impact application performance. To mitigate this risk, asynchronous logging can be an effective solution. We will dive into asynchronous logging a bit later in this chapter.

Understanding and effectively managing the outputs in JVM logging can help you take full advantage of the flexibility and power of the logging system. Also, given that the JVM does not automatically handle log rotation, managing large log files becomes crucial for long-running applications. Without proper log management, log files can grow significantly, consuming disk space and potentially affecting system performance. Implementing log rotation, through either external tools or custom scripts, can help maintain log file sizes and prevent potential issues. Effective log output management not only makes application debugging and monitoring tasks more manageable, but also ensures that logging practices align with the performance and maintenance requirements of your application.

Practical Examples of Using the Unified Logging System

To demonstrate the versatility of the unified logging system, let's start by understanding how to configure it. A comprehensive grasp of this system is crucial for Java developers, particularly for efficient debugging and performance optimization. The `-Xlog` option, a central component of this system, offers extensive flexibility to tailor logging to specific needs. Let's explore its usage through various scenarios:

- **Configuring the logging system using the `-Xlog` option:** The `-Xlog` option is a powerful tool used to configure the logging system. Its flexible syntax, `-Xlog:tags:output:decorators:level`, allows you to customize what is logged, where it is logged, and how it is presented.
- **Enabling logging for specific tags:** If you want to enable logging for the `gc` and `compiler` tags at the `info` level, you can do so with the following command:

  ```
  $ java -Xlog:gc,compiler:info MyApp
  ```

- **Specifying different levels for different tags:** It's also possible to specify different levels for different tags. For instance, if you want `gc` logs at the `info` level, but `compiler` logs at the `warning` level, you can use this command:

  ```
  $ java -Xlog:gc=info,compiler=warning MyApp
  ```

- **Logging to different outputs:** By default, logs are written to *stdout*. However, you can specify a different output, such as a file. For example:

  ```
  $ java -Xlog:gc:file=gc.log MyApp
  ```

  This will write all `gc` logs to the file `gc.log`.

- **Using decorators to include additional information:** You can use decorators to include more context in your log messages. For instance, to include timestamps with your `gc` logs, you can use this command:

  ```
  $ java -Xlog:gc*:file=gc.log:time MyApp
  ```

  This will write `gc` logs to `gc.log`, with each log message being prefixed with the timestamp.

- **Enabling all logs:** For comprehensive logging, which is particularly useful during debugging or detailed analysis, you can enable all log messages across all levels with the following command:

  ```
  $ java -Xlog:all=trace MyApp
  ```

  This command ensures that every possible log message from the JVM, from the `trace` to `error` level, is captured. While this provides a complete picture of JVM operations, it can result in a large volume of log data, so it should be used judiciously.

- **Disabling logging:** If you want to turn off logging, you can do so with the following command:

  ```
  $ java -Xlog:off MyApp
  ```

Benchmarking and Performance Testing

The unified logging system is crucial in benchmarking and performance testing scenarios. By thoroughly analyzing the logs, developers can understand how their applications perform under different workloads and configurations. This information can be used to fine-tune those applications for optimal performance.

For instance, the garbage collection logs can provide insights into an application's memory usage patterns, which can guide decisions on adjusting heap size and fine-tuning other GC parameters for efficiency. In addition, other types of logs, such as JIT compilation logs, thread activity logs, or the `monitorinflation` logs discussed earlier can contribute valuable information. JIT compilation logs, for example, can help identify performance-critical methods,[3] whereas the `thread` and `monitorinflation` logs can be used to diagnose concurrency issues or inefficiencies in thread utilization.

In addition to utilizing them during performance testing, integrating these logs into regression testing frameworks is vital. It ensures that new changes or updates do not adversely affect application performance. By doing so, continuous integration pipelines can significantly streamline the process of identifying and addressing performance bottlenecks in a continuous delivery environment.

However, it's important to note that interpreting these logs often requires a nuanced understanding of JVM behavior. For instance, GC logs can be complex and necessitate a deep understanding of memory management in Java. Similarly, deciphering JIT compilation logs requires knowledge of how the JVM optimizes code execution. Balancing log analysis with other performance testing methodologies ensures a comprehensive evaluation of application performance.

Tools and Techniques

Analyzing JVM logs can be a complex endeavor due to the potential influx of data into such logs. However, the complexity of this task is significantly reduced with the aid of specialized tools and techniques. Log analysis tools are adept at parsing logs and presenting the data in a more manageable and digestible format. These tools can filter, search, and aggregate log data, making it easier to pinpoint relevant information. Similarly, visualization tools can help with pattern identification and detection of trends within the log data. They can turn textual log data into graphical representations, making it easier to spot anomalies or performance bottlenecks.

Moreover, based on historical data, machine learning techniques can predict future performance trends. This predictive analysis can be crucial in proactive system maintenance and optimization. However, effective use of these techniques requires a substantial body of historical data and a foundational understanding of data science principles.

These tools are not only instrumental in problem identification but also play a role in operational responses, such as triggering alerts or automated actions based on log analysis results. We will learn about performance techniques in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH."

[3] We covered performance-critical methods in Chapter 1: "The Performance Evolution of Java: The Language and the Virtual Machine."

Optimizing and Managing the Unified Logging System

Although the unified logging system is an indispensable tool for diagnostics and monitoring, its use demands careful consideration, particularly in regard to applications' performance. Excessive logging can lead to performance degradation, as the system spends additional resources writing to the log files. Moreover, extensive logging can consume significant amounts of disk space, which could be a concern in environments with limited storage. To help manage disk space usage, log files can be compressed, rotated, or moved to secondary storage as necessary.

Thus, balancing the need for detailed logging information and the potential performance impacts is crucial. One approach to do so is to adjust the logging level to include only the most important messages in production environments. For instance, logging some of the messages at the `error` and `warning` levels might be more appropriate than logging all messages at the `info`, `debug`, or `trace` level in high-traffic scenarios.

Another consideration is the output destination for the log messages. During the application's development stages, it's recommended that log messages be directly output on *stdout*—which is why it's the default for the logging system. However, writing log messages to a file might be more suitable in a production environment, so that these messages can be archived and analyzed later as needed.

One advanced feature of the unified logging system is its ability to adjust logging levels at runtime dynamically. This capability is particularly beneficial for troubleshooting issues in live applications. For example, you might typically run your application with standard logging to maximize performance. However, during traffic surges or when you observe an increase in specific patterns, you can temporarily change the logging tag levels to gather more relevant information. Once the issue is resolved and normalcy is restored, you can revert to the original logging level without restarting the application.

This dynamic logging feature is facilitated by the `jcmd` utility, which allows you to send commands to a running JVM. For instance, to increase the logging level for the `gc*` tag to `trace`, and to add some useful decorators, you can use the following command:

```
$ jcmd <pid> VM.log what=gc*=trace decorators=uptimemillis,tid,hostname
```

Here, `<pid>` is the process ID of the running JVM. After executing this command, the JVM will start logging all garbage collection activities at the `trace` level, including the uptime in milliseconds, thread ID, and hostname decorators. This provides us with comprehensive diagnostic information.

To explore the full range of configurable options, you can use the following command:

```
$ jcmd <pid> help VM.log
```

Although the unified logging system provides a wealth of information, it's essential to use it wisely. You can effectively manage the trade-off between logging detail and system performance by adjusting the logging level, choosing an appropriate output, and leveraging the dynamic logging feature.

Asynchronous Logging and the Unified Logging System

The evolution of Java's unified logging system has resulted in another notable advanced (optional) feature: asynchronous logging. Unlike traditional synchronous logging, which (under certain conditions) can become a bottleneck in application performance, asynchronous logging delegates log-writing tasks to a separate thread. Log entries are placed in a staging buffer before being transferred to the final output. This method minimizes the impacts of logging on the main application threads—a consideration that is crucial for modern, large-scale, and latency-sensitive applications.

Benefits of Asynchronous Logging

- **Reduced latency:** By offloading logging activities, asynchronous logging significantly lowers the latency in application processing, ensuring that key operations aren't delayed by log writing.
- **Improved throughput:** In I/O-intensive environments, asynchronous logging enhances application throughput by handling log activities in parallel, freeing up the main application to handle more requests.
- **Consistency under load:** One of the most significant advantages of asynchronous logging is its ability to maintain consistent performance, even under substantial application loads. It effectively mitigates the risk of logging-induced bottlenecks, ensuring that performance remains stable and predictable.

Implementing Asynchronous Logging in Java

Using Existing Libraries and Frameworks

Java offers several robust logging frameworks equipped with asynchronous capabilities, such as Log4j2 and Logback:

- **Log4j2[4]:** Log4j2's asynchronous logger leverages the LMAX Disruptor, a high-performance interthread messaging library, to offer low-latency and high-throughput logging. Its configuration involves specifying *AsyncLogger* in the configuration file, which then directs the logging events to a ring buffer, reducing the logging overhead.
- **Logback[5]:** In Logback, asynchronous logging can be enabled through the *AsyncAppender* wrapper, which buffers log events and dispatches them to the designated appender. Configuration entails wrapping an existing appender with *AsyncAppender* and tweaking the buffer size and other parameters to balance performance and resource use.

In both frameworks, configuring the asynchronous logger typically involves XML or JSON configuration files, where various parameters such as buffer size, blocking behavior, and underlying appenders can be defined.

[4] https://logging.apache.org/log4j/2.x/manual/async.html

[5] https://logback.qos.ch/manual/appenders.html#AsyncAppender

Custom Asynchronous Logging Solutions

In some scenarios, the standard frameworks may not meet the unique requirements of an application. In such cases, developing a custom asynchronous logging solution may be appropriate. Key considerations for a custom implementation include

- **Thread safety:** Ensuring that the logging operations are thread-safe to avoid data corruption or inconsistencies.
- **Memory management:** Efficiently handling memory within the logging process, particularly in managing the log message buffer, to prevent memory leaks or high memory consumption.
- **Example implementation:** A basic custom asynchronous logger might involve a producer–consumer pattern, where the main application threads (producers) enqueue log messages to a shared buffer, and a separate logging thread (consumer) dequeues and processes these messages.

Integration with Unified JVM Logging

- **Challenges and solutions:** For JVM versions prior to JDK 17 or in complex logging scenarios, integrating external asynchronous logging frameworks with the unified JVM logging system requires careful configuration to ensure compatibility and performance. This might involve setting JVM flags and parameters to correctly route log messages to the asynchronous logging system.
- **Native support in JDK 17 and beyond:** Starting from JDK 17, the unified logging system natively supports asynchronous logging. This can be enabled using the `-Xlog:async` option, allowing the JVM's internal logging to be handled asynchronously without affecting the performance of the JVM itself.

Best Practices and Considerations

Managing Log Backpressure

- **Challenge:** In asynchronous logging, backpressure occurs when the rate of log message generation exceeds the capacity for processing and writing these messages. This can lead to performance degradation or loss of log data.
- **Bounded queues:** Implementing bounded queues in the logging system can help manage the accumulation of log messages. By limiting the number of unprocessed messages, these queues prevent memory overutilization.
- **Discarding policies:** Establishing policies for discarding log messages when queues reach their capacity can prevent system overload. This might involve dropping less critical log messages or summarizing message content.
- **Real-life examples:** A high-traffic web application might implement a discarding policy that prioritizes error logs over informational logs during peak load times to maintain system stability.

Ensuring Log Reliability

- **Criticality:** The reliability of a logging system is paramount, particularly during application crashes or abrupt shutdowns. Loss of log data in such scenarios can hinder troubleshooting and impact compliance.
- **Write-ahead logs:** Implementing write-ahead logging ensures that log data is preserved even if the application fails. This technique involves writing log data to a persistent storage site before the actual logging operation concludes.
- **Graceful shutdown procedures**: Designing logging systems to handle shutdowns gracefully ensures that all buffered log data is written out before the application stops completely.
- **Strategies in adverse conditions:** Include redundancy in the logging infrastructure to capture log data even if the primary logging mechanism fails. This is particularly relevant in distributed systems where multiple components can generate logs.

Performance Tuning of Asynchronous Logging Systems

- **Tuning for optimal performance**
  - **Thread pool sizes:** Adjusting the number of threads dedicated to asynchronous logging is crucial. More threads can handle higher log volumes, but excessive threads may cause heavy CPU overhead.
  - **Buffer capacities:** If your application experiences periodic spikes in log generation, increasing the buffer size can help absorb these spikes.
  - **Processing intervals:** The frequency at which the log is flushed from the buffer to the output can impact both performance and log timeliness. Adjusting this interval helps balance CPU usage against the immediacy of log writing.
- **JDK 17 considerations with asynchronous logging**
  - **Buffer size tuning:** The `AsyncLogBufferSize` parameter is crucial for managing the volume of log messages that the system can handle before flushing. A larger buffer can accommodate more log messages, which is beneficial during spikes in log generation. However, a larger buffer also requires more memory.
  - **Monitoring JDK 17's asynchronous logging:** With the new asynchronous logging feature, monitoring tools and techniques might need to be updated to track the utilization of the new buffer and the performance of the logging system.
- **Monitoring and metrics**
  - **Buffer utilization:** You could use a logging framework's internal metrics or a custom monitoring script to track buffer usage. If the buffer consistently has greater than 80% utilization, that's a sign you should increase its size.
  - **Queue lengths:** Monitoring tools or logging framework APIs could be used to track the length of the logging queue. Persistently long queues require more logging threads or a larger buffer.

- **Write latencies:** Measuring the time difference between log message creation and its final write operation could reveal write latencies. Optimizing file I/O operations or distributing logs across multiple files or disks could help if these latencies are consistently high.
- **JVM tools and metrics:** Tools like JVisualVM[6] and Java Mission Control (JMC)[7] can provide insights into JVM performance, including aspects that might impact logging, such as thread activity or memory usage.

Understanding the Enhancements in JDK 11 and JDK 17

The unified logging system was first introduced in JDK 9 and has since undergone refinements and enhancements in subsequent versions of the JDK. This section discusses the significant improvements made in JDK 11 and JDK 17 with respect to unified logging.

JDK 11

In JDK 11, one of the notable improvements to the unified logging framework was addition of the ability to dynamically configure logging at runtime. This enhancement was facilitated by the `jcmd` utility, which allows developers to send commands to a running JVM, including commands to adjust the log level. This innovation has significantly benefited developers by enabling them to increase or decrease logging, depending on the diagnostic requirements, without restarting the JVM.

JDK 11 also improved `java.util.logging` by introducing new methods for Logger and LogManager. These additions further enhanced logging control and capabilities, providing developers with more granularity and flexibility when working with Java logs.

JDK 17

JDK 17 introduced native support for asynchronous logging improvements within the unified logging system of the JVM. In addition, JDK 17 brought broader, indirect improvements to the logging system's overall performance, security, and stability. Such improvements contribute to the effectiveness of unified logging, as a more stable and secure system is always beneficial for accurate and reliable logging.

Conclusion

In this chapter, we explored the unified logging system, showcasing its practical usage and outlining the enhancements introduced in subsequent releases of the JDK. The unified logging system in JVM provides a single, consistent interface for all log messages. This harmony across the JVM has enhanced both readability and parseability, effectively eliminating issues with interleaved or interspersed logs. Preserving the ability to redirect logs to a file—a feature available even prior to JDK 9—was an essential part of retaining the usefulness of the earlier logging mechanisms. The logs continue to be human-readable, and timestamps, by default, reflect the uptime.

[6] https://visualvm.github.io/

[7] https://jdk.java.net/jmc/8/

The unified logging system also offers significant value-added features. The introduction of dynamic logging in JDK 11 has allowed developers to make adjustments to logging commands during runtime. Combined with the enhanced visibility and control over the application process provided by the unified logging system, developers can now specify the depth of log information required via command-line inputs, improving the efficiency of debugging and performance testing.

The information obtained from the unified logging system can be used to tune JVM parameters for optimal performance. For example, I have my own GC parsing and plotting toolkit that is now more consolidated and streamlined to provide insights into the memory usage patterns, pause events, heap utilization, pause details, and various other patterns associated with the collector type. This information can be used to adjust the heap size, choose the appropriate garbage collector, and tune other garbage collection parameters.

Newer features of Java, such as Project Loom and Z Garbage Collector (ZGC), have significant implications for JVM performance. These features can be monitored via the unified logging system to understand their impacts on application performance. For instance, ZGC logs can provide insights into the GC's pause times and efficiency in reclaiming memory. We will learn more about these features in upcoming chapters.

In conclusion, the unified logging system in JVM is a potent tool for developers. By understanding and leveraging this system, developers can streamline their troubleshooting efforts, gaining deeper insights into the behavior and performance of their Java applications. Continuous enhancements across different JDK versions have made the unified logging system even more robust and versatile, solidifying its role as an essential tool in the Java developers' toolkit.

Chapter 5

End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH

Introduction

Welcome to a pivotal stage in our journey through Java Virtual Machine (JVM) performance engineering. This chapter marks a significant transition, where we move from understanding the historical context and foundational structures of Java to applying this knowledge to optimize performance. This sets the stage for our upcoming in-depth exploration of performance optimization in the OpenJDK HotSpot JVM.

Together, we've traced the evolution of Java, its type system, and the JVM, witnessing Java's transition from a monolithic structure to a modular one. Now, we're poised to apply these concepts to the vital task of performance optimization.

Performance is a cornerstone of any software application's success. It directly influences the user experience, determining how swiftly and seamlessly an application responds to user interactions. The JVM, the runtime environment where Java applications are executed, is central to this performance. By learning how to adjust and optimize the JVM, you can significantly boost your Java application's speed, responsiveness, and overall performance. This could involve strategies such as adjusting memory and garbage collection patterns, using optimized code or libraries, or leveraging JVM's just-in-time (JIT) compilation capabilities and thresholds to your advantage. Mastering these JVM optimization techniques is a valuable skill for enhancing your Java applications' performance.

Building on our previous foundational investigation into the unified JVM logging interface, we now can explore performance metrics and consider how to measure and interpret them effectively. As we navigate the intricacies of JVM, we'll examine its interaction with hardware, the

role of memory barriers, and the use of volatile variables. We'll also delve into the comprehensive process of performance optimization, which encompasses monitoring, profiling, analysis, and tuning.

This chapter transcends mere theoretical discourse, embodying a distillation of over two decades of hands-on experience in performance optimization. Here, indispensable tools for Java performance measurement are introduced, alongside diagnostic and analysis methodologies that have been meticulously cultivated and refined throughout the years. This approach, involving a detailed investigation of subsystems to pinpoint potential issues, offers a crucial roadmap for software developers with a keen focus on performance optimization.

My journey has led to a deep understanding of the profound impact that underlying hardware and software stack layers have on an application's performance, hence this chapter sheds light on the interplay between hardware and software. Another dedicated section recognizes the critical importance of benchmarking, whether for assessing the impact of feature releases or analyzing performance across different system layers. Together, we will explore practical applications using the Java Micro-benchmark Harness (JMH). Additionally, we will look at the built-in profilers within JMH, such as *perfasm*, a powerful tool that can unveil the low-level performance characteristics of your code.

This chapter, therefore, is more than just a collection of theories and concepts. It's a practical guide, filled with examples and best practices, all drawn from extensive field experience. Designed to bridge the gap between theory and practice, it serves as a springboard for the detailed performance optimization discussions in the subsequent chapters of this book, ensuring readers are well-equipped to embark on the advanced performance engineering journey.

Performance Engineering: A Central Pillar of Software Engineering

Decoding the Layers of Software Engineering

Software engineering is a comprehensive discipline that encapsulates two distinct yet intertwined dimensions: (1) software design and development and (2) software architectural requirements.

Software design and development is the first dimension that we encounter on this journey. It involves crafting a detailed blueprint or schema for a system, which addresses the system's architecture, components, interfaces, and other distinguishing features. These design considerations build the foundation for the functional requirements—the rules that define what the system should accomplish. The functional requirements encapsulate the business rules with which the system must comply, the data manipulations it should perform, and the interactions it should facilitate.

In contrast, software architectural requirements focus on the nonfunctional or quality attributes of the system—often referred to as the "ilities." These requirements outline how the system should behave, encompassing performance, usability, security, and other crucial attributes.

Understanding and striking a balance between these two dimensions—catering to the functional needs while ensuring high-quality attributes—is a critical aspect of software

engineering. With this foundational understanding in place, let's focus on one of these essential qualities—performance.

Performance: A Key Quality Attribute

Among the various architectural requirements, performance holds a distinctive position. It measures how effectively a system—referred to as the "system under test" (SUT)— executes a specific task or a "unit of work" (UoW). Performance is not merely a nice-to-have feature; it's a fundamental quality attribute. It influences both the efficiency of operations and the user experience, and it plays a pivotal role in determining user satisfaction.

To fully comprehend the concept of performance, we must understand the various system components and their impacts on performance. In the context of a managed runtime system like the JVM, a key component that significantly affects performance is the garbage collector (GC). I define the unit of work for GC in performance parlance as "GC work," referring to the specific tasks each GC must perform to meet its performance requirements. For example, a type of GC that I categorize as "throughput-maximizing," exemplified by the Parallel GC in OpenJDK HotSpot VM, focuses on "parallel work." In contrast, a "low-latency-optimizing" collector, as I define it, requires greater control over its tasks to guarantee sub-millisecond pauses. Hence, understanding the individual attributes and contributions of these components is critical, as together they create a holistic picture of system performance.

Understanding and Evaluating Performance

Performance evaluation isn't a mere measure of how fast or slow a system operates. Instead, it involves understanding the interactions between the SUT and the UoW. The UoW could be a single user request, a batch of requests, or any task the system needs to perform. Performance depends on how effectively and efficiently the SUT can execute the UoW under various conditions. It encompasses factors such as the system's resource utilization, throughput, and response time, as well as how these parameters change as the workload increases.

Evaluating performance also requires engaging with the system's users, including understanding what makes them happy and what causes them dissatisfaction. Their insights can reveal hidden performance issues or suggest potential improvements. Performance isn't just about speed—it's about ensuring the system consistently meets user expectations.

Defining Quality of Service

Quality of service (QoS) is a broad term that represents the user's perspective on the service's performance and availability. In today's dynamic software landscape, aligning QoS with service level agreements (SLAs) and incorporating modern site reliability engineering (SRE) concepts like service level objectives (SLOs) and service level indicators (SLIs) is crucial. These elements work together to ensure a shared understanding of what constitutes satisfactory performance.

SLAs define the expected level of service, typically including metrics for system performance and availability. They serve as formal agreements between service providers and users

regarding the expected performance standards, providing a benchmark against which the system's performance can be measured and evaluated.

SLOs are specific, measurable goals within SLAs that reflect the desired level of service performance. They act as targets for reliability and availability, guiding teams in maintaining optimal service levels. In contrast, SLIs are the quantifiable measures used to evaluate the performance of the service against the SLOs. They provide real-time data on various aspects of the service, such as latency, error rates, or uptime, allowing for continuous monitoring and improvement.

Defining clear, quantifiable QoS metrics, in light of SLAs, SLOs, and SLIs, ensures that all parties involved have a comprehensive and updated understanding of service performance expectations and metrics used to gauge them. This approach is increasingly considered best practice in managing application reliability and is integral to modern software development and maintenance.

In refining our understanding of QoS, I have incorporated insights from industry expert Stefano Doni, who is the co-founder and chief technology officer at Akamas.[1] Stefano's suggestion to incorporate SRE concepts like SLOs and SLIs adds significant depth and relevance to our discussion.

Success Criteria for Performance Requirements

Successful performance isn't a "one size fits all" concept. Instead, it should be defined in terms of measurable, realistic criteria that reflect the unique requirements and constraints of each system. These criteria may include metrics such as response time, system throughput, resource utilization, and more. Regular monitoring and measurement against these criteria can help ensure the system continues to meet its performance objectives.

In the next sections, we'll explore the specific metrics for measuring Java performance, highlighting the significant role of hardware in shaping its efficiency. This will lay the groundwork for an in-depth discussion on concurrency, parallelism in hardware, and the characteristics of weakly ordered systems, leading seamlessly into the world of benchmarking.

Metrics for Measuring Java Performance

Performance metrics are critical to understanding the behavior of your application under different conditions. These metrics provide a quantitative basis for measuring the success of your performance optimization efforts. In the context of Java performance optimization, a wide array of metrics are employed to gauge different aspects of applications' behavior and efficiency. These metrics encompass everything from latency, which is particularly insightful when viewed in the context of stimulus-response, to scalability, which assesses the application's ability to handle increasing loads. Efficiency, error rates, and resource consumption are other critical metrics that offer insights into the overall health and performance of Java applications.

[1] Akamas focuses on AI-powered autonomous performance optimization solutions; www.akamas.io/about.

Each metric provides a unique lens through which the performance of a Java application can be examined and optimized. For instance, latency metrics are crucial in real-time processing environments where timely responses are desired. Meanwhile, scalability metrics are vital not just in assessing an application's current capacity to grow and manage peak loads, but also for determining its scaling factor—a measure of how effectively the application can expand its capacity in response to increased demands. This scaling factor is integral to capacity planning because it helps predict the necessary resources and adjustments needed to sustain and support the application's growth. By understanding both the present performance and potential scaling scenarios, you can ensure that the application remains robust and efficient, adapting seamlessly to both current and future operational challenges.

A particularly insightful aspect of evaluating responsiveness and latency is understanding the call chain depth in an application. Every stage in a call chain can introduce its own delay, and these delays can accumulate and amplify as they propagate down the chain. This effect means that an issue at an early stage can significantly exacerbate the overall latency, affecting the application's end-to-end responsiveness. Such dynamics are crucial to understand, especially in complex applications with multiple interacting components.

In the Java ecosystem, these metrics are not only used to monitor and optimize applications in production environments, but they also play a pivotal role during the development and testing phases. Tools and frameworks within the Java world, such as JMH, offer specialized functionalities to measure and analyze these performance indicators.

For the purpose of this book, while acknowledging the diversity and importance of various performance metrics in Java, we will concentrate on four key areas: footprint, responsiveness, throughput, and availability. These metrics have been chosen due to their broad applicability and impact on Java applications:

Footprint

Footprint pertains to the space and resources required to run your software. It's a crucial factor, especially in environments with limited resources. The footprint comprises several aspects:

- **Memory footprint:** The amount of memory your software consumes during execution. This includes both the Java heap for object storage and native memory used by the JVM itself. The memory footprint encompasses runtime data structures, which constitute the application system footprint. Native memory tracking (NMT) in the JVM can play a key role here, providing detailed insights into the allocation and usage of native memory. This is essential for diagnosing potential memory inefficiencies or leaks outside of the Java heap. In Java applications, optimizing memory allocation and garbage collection strategies is key to managing this footprint effectively.
- **Code footprint:** The size of the actual executable code of your software. Large codebases can require more memory and storage resources, impacting the system's overall footprint. In managed runtime environments, such as those provided by the JVM, the code footprint can have direct impact on the process, influencing overall performance and resource utilization.

- **Physical resources:** The use of resources such as storage, network bandwidth, and, especially, CPU usage. CPU footprint, or the amount of CPU resources consumed by your application, is a critical factor in performance and cost-effectiveness, especially in cloud environments. In the context of Java applications, the choice of GC algorithms can significantly affect CPU usage. Stefano notes that appropriate optimization of the GC algorithm can sometimes result in halving CPU usage, leading to considerable savings in cloud-based deployments.

These aspects of the footprint can significantly influence your system's efficiency and operational costs. For instance, a larger footprint can lead to longer start-up times, which is critical in contexts like container-based and serverless architectures. In such environments, each container includes the application and its dependencies, dictating a specific footprint. If this footprint is large, it can lead to inefficiencies, particularly in scenarios when multiple containers are running on the same host. Such situations can sometimes give rise to "noisy neighbor" issues, where one container's resource-intensive operations monopolize system resources, adversely affecting the performance of others. Hence, optimizing the footprint is not just about improving individual container efficiency; it is also about managing resource contention and ensuring more efficient resource usage on shared platforms.

In serverless computing, although resource management is automatic, an application with a large footprint may experience longer start-up times, known as "cold starts." This can impact the responsiveness of serverless functions, so footprint optimization is crucial for better performance. Similarly, for most Java applications, the memory footprint often directly correlates with the workload of the GC, which can impact performance. This is particularly true in cloud environments and with microservices architecture, where resource consumption directly affects responsiveness and operational overhead and requires careful capacity planning.

Monitoring Non-Heap Memory with Native Memory Tracking

Building on our earlier discussion on NMT in the JVM, let's take a look at how NMT can be effectively employed to monitor non-heap memory. NMT provides valuable options for this purpose: a summary view—offering an overview of aggregate usage, and a detailed view—presenting a breakdown of usage by individual call sites:

```
-XX:NativeMemoryTracking=[off | summary | detail]
```

NOTE Enabling NMT, especially at the `detail` level, may introduce some performance overhead. Therefore, it should be used judiciously, particularly in production environments.

For real-time insights, `jcmd` can be used to grab the runtime `summary` or `detail`. For instance, to retrieve a `summary`, the following command is used:

```
$ jcmd <pid> VM.native_memory summary
<pid>:
Native Memory Tracking:
(Omitting categories weighting less than 1KB)
Total: reserved=12825060KB, committed=11494392KB
-                 Java Heap (reserved=10485760KB, committed=10485760KB)
                            (mmap: reserved=10485760KB, committed=10485760KB)
-                     Class (reserved=1049289KB, committed=3081KB)
                            (classes #4721)
                            (  instance classes #4406, array classes #315)
                            (malloc=713KB #12928)
                            (mmap: reserved=1048576KB, committed=2368KB)
                            (  Metadata:   )
                            (    reserved=65536KB, committed=15744KB)
                            (    used=15522KB)
                            (    waste=222KB =1.41%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=2368KB)
                            (    used=2130KB)
                            (    waste=238KB =10.06%)
-                    Thread (reserved=468192KB, committed=468192KB)
                            (thread #456)
                            (stack: reserved=466944KB, committed=466944KB)
                            (malloc=716KB #2746)
                            (arena=532KB #910)
-                      Code (reserved=248759KB, committed=18299KB)
                            (malloc=1075KB #5477)
                            (mmap: reserved=247684KB, committed=17224KB)
-                        GC (reserved=464274KB, committed=464274KB)
                            (malloc=41894KB #10407)
                            (mmap: reserved=422380KB, committed=422380KB)
-                  Compiler (reserved=543KB, committed=543KB)
                            (malloc=379KB #424)
                            (arena=165KB #5)
-                  Internal (reserved=1172KB, committed=1172KB)
                            (malloc=1140KB #12338)
                            (mmap: reserved=32KB, committed=32KB)
-                     Other (reserved=818KB, committed=818KB)
                            (malloc=818KB #103)
-                    Symbol (reserved=3607KB, committed=3607KB)
                            (malloc=2959KB #79293)
                            (arena=648KB #1)
```

```
-    Native Memory Tracking (reserved=2720KB, committed=2720KB)
                            (malloc=352KB #5012)
                            (tracking overhead=2368KB)
-        Shared class space (reserved=16384KB, committed=12176KB)
                            (mmap: reserved=16384KB, committed=12176KB)
-               Arena Chunk (reserved=16382KB, committed=16382KB)
                            (malloc=16382KB)
-                   Logging (reserved=7KB, committed=7KB)
                            (malloc=7KB #287)
-                 Arguments (reserved=2KB, committed=2KB)
                            (malloc=2KB #53)
-                    Module (reserved=252KB, committed=252KB)
                            (malloc=252KB #1226)
-                 Safepoint (reserved=8KB, committed=8KB)
                            (mmap: reserved=8KB, committed=8KB)
-           Synchronization (reserved=1195KB, committed=1195KB)
                            (malloc=1195KB #19541)
-            Serviceability (reserved=1KB, committed=1KB)
                            (malloc=1KB #14)
-                Metaspace (reserved=65693KB, committed=15901KB)
                            (malloc=157KB #238)
                            (mmap: reserved=65536KB, committed=15744KB)
-      String Deduplication (reserved=1KB, committed=1KB)
                            (malloc=1KB #8)
```

As you can see, this command generates a comprehensive report, categorizing memory usage across various JVM components such as the *Java Heap*, *Class*, *Thread*, *GC*, and *Compiler* areas, among others. The output also includes memory reserved and committed, aiding in understanding how memory is allocated and utilized by different JVM subsystems. You can also get diffs for either the `detail` level or `summary` view.

Mitigating Footprint Issues

To address footprint-related challenges, insights gained from NMT, GC logs, JIT compilation data, and other JVM unified logging outputs can inform various mitigation strategies. Key approaches include

- **Data structure optimization:** Prioritize memory-efficient data structures by analyzing usage and selecting lighter, space-efficient alternatives tailored to your application's predictable data patterns. This approach ensures optimal memory utilization.
- **Code refactoring:** Streamline code to reduce redundant memory operations and refine algorithmic efficiency. Minimizing object creation, preferring primitives over boxed types, and optimizing string handling are key strategies to reduce memory usage.

- **Adaptive sizing and GC strategy:** Leverage modern GCs' adaptive sizing features to dynamically adjust to your application's behavior and workload. Carefully selecting the right GC algorithm—based on your footprint needs—enhances runtime efficiency. Where supported, incorporate `-XX:SoftMaxHeapSize` for applications with specific performance targets, allowing controlled heap expansion and optimizing GC pause times. This strategy, coupled with judiciously setting initial (`-Xms`) and maximum (`-Xmx`) heap sizes, forms a comprehensive approach to managing memory efficiently.

These techniques when applied cohesively, ensure a balanced approach to resource management, enhancing performance across various environments.

Responsiveness

Latency and responsiveness, while related, illuminate different aspects of system performance. Latency measures the time elapsed from receiving a stimulus (like a user request or system event) to when a response is generated and delivered. It's a critical metric, but it captures only one dimension of performance.

Responsiveness, on the other hand, extends beyond mere speed of reaction. It evaluates how a system adapts under varied load conditions and sustains performance over time, offering a holistic view of the system's agility and operational efficiency. It encompasses the system's ability to promptly react to inputs and execute the necessary operations to produce a response.

In assessing system responsiveness, the previously discussed concepts of SLOs and SLIs become particularly relevant. By setting clear SLOs for response times, you establish the performance benchmarks your system aims to achieve. Consider the following essential questions to ask when assessing responsiveness:

- What is the targeted response time?
- Under what conditions is it acceptable for response times to exceed the targeted benchmarks, and to what extent is this deviation permissible?
- How long can the system endure delays?
- Where is response time measured: client side, server side, or end-to-end?

Establishing SLIs in response to these questions sets clear, quantifiable benchmarks to measure actual system performance relative to your SLOs. Tracking these indicators help identify any deviations and guide your optimization efforts. Tools such as JMeter, LoadRunner, and custom scripts can be used to measure responsiveness under different load conditions and perform continuous monitoring of the SLIs, providing insights into how well your system maintains the performance standards set by your SLOs.

Throughput

Throughput is another critical metric that quantifies how many operations your system can handle per unit of time. It is vital in environments that require processing of high volumes of transactions or data. For instance, my role in performance consulting for a significant e-commerce platform highlighted the importance of maximizing throughput during peak periods like holiday sales. These periods could lead to significant spikes in traffic, so the system

needed to be able to scale resources to maintain high throughput levels and "keep the 4-9s in tail latencies" (that is, complete 99.99% of all requests within a specified time frame). By doing so, the platform could provide an enhanced user experience and ensure successful business operations.

Understanding your system's capacity to scale is crucial in these situations. Scaling strategies can involve horizontal scaling (that is, scaling out), which involves adding more systems to distribute the load, and vertical scaling (that is, scaling up), where you augment the capabilities of existing systems. For instance, in the context of the e-commerce platform during the holiday sales event, employing an effective scaling strategy is essential. Cloud environments often provide the flexibility to employ such strategies effectively, allowing for dynamic resource allocation that adapts to varying demand.

Availability

Availability is a measure of your system's uptime—essentially, how often it is functional and accessible. Measures such as mean time between failure (MTBF) are commonly used to gauge availability. High availability is often essential for maintaining user satisfaction and meeting SLAs. Other important measures include mean time to recovery (MTTR), which indicates the average time to recover from a failure, and redundancy measures, such as the number of failover instances in a cluster. These can provide additional insights into the resilience of your system.

In my role as a performance consultant for a large-scale, distributed database company, high availability was a major focus area. The company specializes in developing scale-out architectures for data warehousing and machine learning, utilizing Hadoop for handling vast data sets. Achieving high availability requires robust fallback systems and strategies for scalability and redundancy. For example, in our Hadoop-based systems, we ensured high availability by maintaining multiple failover instances within a cluster. This setup allowed seamless transition to a backup instance in case of failure, minimizing downtime. Such redundancy is critical not only for continuous service but also for sustaining performance metrics like high throughput during peak loads, ensuring the system manages large volumes of requests or data without performance degradation.

Moreover, high availability is essential for maintaining consistent responsiveness, enabling swift recovery from failures and keeping response times stable, even under adverse conditions. In database systems, enhancing availability means increasing failover instances across clusters and diversifying them for greater redundancy. This strategy, rigorously tested, ensures smooth, automatic failover during outages, maintaining uninterrupted service and user trust.

Particularly in large-scale systems like those based on the Hadoop ecosystem—which emphasizes components like HDFS for data storage, YARN for resource management, or HBase for NoSQL database capabilities—understanding and planning for fallback systems is critical. Downtime in such systems can significantly impact business operations. These systems are designed for resilience, seamlessly taking over operations to maintain continuous service, integral to user satisfaction.

Through my experiences, I have learned that achieving high availability is a complex balancing act that requires careful planning, robust system design, and ongoing monitoring to ensure system resilience and maintain the trust of your users. Cloud deployments can significantly aid in achieving this balance. With their built-in automated failover and redundancy procedures, cloud services fortify systems against unexpected downtimes. This adaptability, along with the flexibility to scale resources on demand, is crucial during peak load periods, ensuring high availability under various conditions.

In conclusion, achieving high availability transcends technical execution; it's a commitment to building resilient systems that reliably meet and exceed user expectations. This dedication, pivotal in our digital era, demands not just advanced solutions like cloud deployments but also a holistic approach to system architecture. It involves integrating strategic design principles that prioritize seamless service continuity, reflecting a comprehensive commitment to reliability and performance excellence.

Digging Deeper into Response Time and Availability

SLAs are essential in setting benchmarks for response time. They might include metrics such as average response time, maximum (worst-case) response time, 99th percentile response time, and other high percentiles (commonly referred to as the "4-9s" and "5-9s").

A visual representation of response times across different systems can reveal significant insights. In particular, capturing the worst-case scenario can identify potential availability issues that might go unnoticed when looking at average or best-case scenarios. As an example, consider Table 5.1, which presents recorded response times across four different systems.

Table 5.1 **Example: Monitoring Response Times**

	Number of GC Events	Minimum (ms)	Average (ms)	99th Percentile (ms)	Maximum (ms)
System 1	36,562	7.622	310.415	945.901	2932.333
System 2	34,920	7.258	320.778	1006.677	2744.587
System 3	36,270	6.432	321.483	1004.183	1681.305
System 4	40,636	7.353	325.670	1041.272	18,598.557

The initial examination of the minimum, average, and 99th percentile response times in Table 5.1 might not raise any immediate concerns. However, the maximum response time for System 4, revealing a pause as long as ~18.6 seconds, warrants a deeper investigation. Initially, such extended pause times might suggest GC inefficiencies, but they could also stem from broader system latency issues, potentially triggering failover mechanisms in highly available distributed systems. Understanding this scenario is critical, as frequent occurrences can severely impact the MTBF and MTTR, pushing these metrics beyond acceptable thresholds.

This example underlines the importance of capturing and evaluating worst-case scenarios when assessing response times. Tools that can help profile and analyze such performance issues include system and application profilers like Intel's VTune, Oracle's Performance Analyzer, Linux *perf*, VisualVM,[2] and Java Mission Control,[3] among others. Utilizing these tools is instrumental in diagnosing systemic performance bottlenecks that compromise responsiveness, enabling targeted optimizations to enhance system availability.

The Mechanics of Response Time with an Application Timeline

To better understand response time, consider Figure 5.1, which depicts an application timeline. The timeline illustrates two key events: the arrival of a stimulus S_0 at time T_0, and the application sending back a response R_0 at time T_1. Upon receiving the stimulus, the application continues to work (A_{cw}) to process this new stimulus, and then sends back a response R_0 at time T_1. The work happening before the stimulus arrival is denoted as A_w. Thus, Figure 5.1 illustrates the concept of response time in the context of application work.

Understanding Response Time and Throughput in the Context of Pauses

Stop-the-world (STW) pauses refer to periods when all application threads are suspended. These pauses, which are required for garbage collection, can interrupt application work and impact response time and throughput. Considering the application timeline in the context of STW pauses can help us better appreciate how response time and throughput are affected by such events.

Let's consider a scenario where STW events interrupt the application's continuous work (A_{cw}). In Figure 5.2, two portions of the uninterrupted A_{cw}—A_{cw_0} and A_{cw_1}—are separated by STW events. In this altered timeline, the response R_0 that was supposed to be sent at time T_1 in the uninterrupted scenario is now delayed until time T_2. This results in an elongated response time.

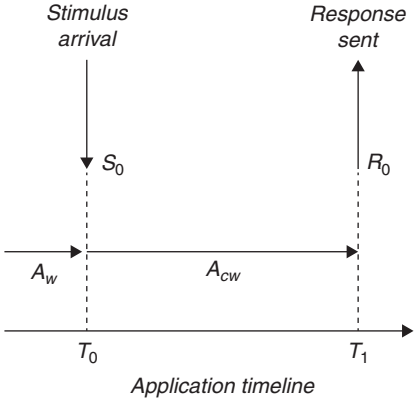

Figure 5.1 An Application Timeline Showing the Arrival of a Stimulus and the Departure of the Response

[2] https://visualvm.github.io/

[3] https://jdk.java.net/jmc/8/

By comparing the two timelines (uninterrupted and interrupted) side by side, as shown in Figure 5.3, we can quantify the impact of these STW events. Let's focus on two key metrics: the total work done in both scenarios within the same period and the throughput.

1. Work done:
   - In the uninterrupted scenario, the total work done is A_{cw}.
   - In the interrupted scenario, the total work done is the sum of A_{cw_0} and A_{cw_1}.
2. Throughput (W_d):
   - For the **uninterrupted timeline**, the throughput (W_{da}) is calculated as the total work done (A_{cw}) divided by the duration from stimulus arrival to response dispatch ($T_1 - T_0$).
   - In the **interrupted timeline**, the throughput (W_{db}) is the sum of the work done in each segment ($A_{cw_0} + A_{cw_1}$) divided by the same duration ($T_1 - T_0$).

Given that the interruptions caused by the STW pauses result in $A_{cw_0} + A_{cw_1}$ being less than A_{cw}, W_{db} will consequently be lower than W_{da}. This effectively illustrates the reduction in throughput during the response window from T_0 to T_1, highlighting how STW events impact the application's efficiency in processing tasks and ultimately prolong the response time.

Through our exploration of response times, throughput, footprint and availability, it's clear that optimizing software is just one facet of enhancing system performance. As we navigate the nuances of software subsystems, we recognize that the components influencing the 'ilities' of a system extend beyond just the software layer. The hardware foundation on which the JVM and applications operate plays an equally pivotal role in shaping overall performance and reliability.

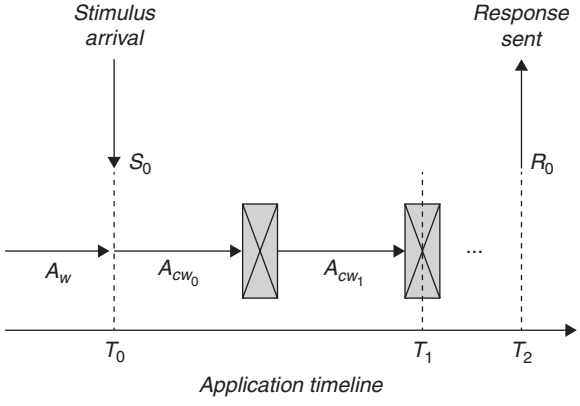

Figure 5.2 An Application Timeline with Stop-the-World Pauses

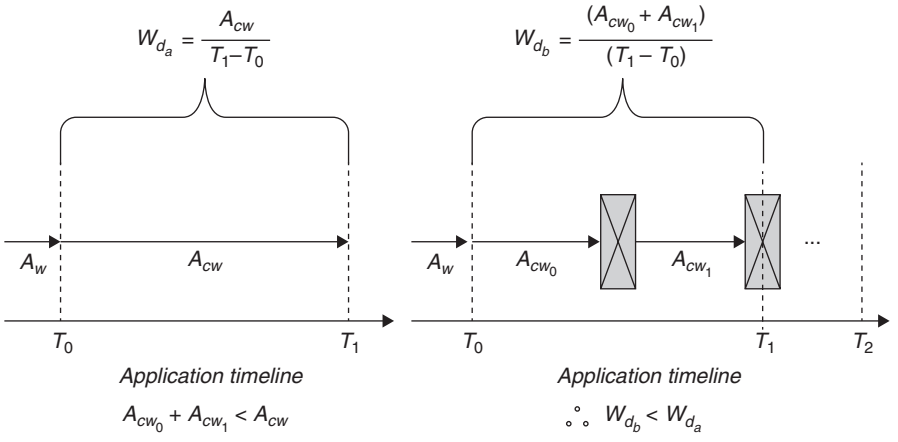

Figure 5.3 Side-by-Side Comparison to Highlight Work Done and Throughput

The Role of Hardware in Performance

In our exploration of JVM performance, we've traversed key software metrics like responsiveness, throughput, and more. Beyond these software intricacies lies a critical dimension: the underlying hardware.

While software innovations capture much attention, hardware's role in shaping performance is equally crucial. As cloud architectures evolve, the strategic importance of hardware becomes more apparent, with vendors optimizing for both power and cost-efficiency.

Considering environments from data centers to the cloud, the impact of hardware changes—such as processor architecture adjustments or adding processing cores—is profound. How these modifications affect performance metrics or system resilience prompts a deeper investigation. Such inquiries not only enhance our understanding of hardware's influence on performance but also guide effective hardware stack modifications, underscoring the need for a balanced approach to software and hardware optimization in pursuit of technological efficiency.

Building on hardware's pivotal role in system performance, let's consider a containerized Java application scenario using Netty, renowned for its high-performance and nonblocking I/O capabilities. While Netty excels in managing concurrent connections, its efficacy extends beyond software optimization to how it integrates with cloud infrastructure and leverages the capabilities of underlying hardware within the constraints of a container environment.

Containerization introduces its own set of performance considerations, including resource isolation, overhead, and the potential for density-related issues within the host system. Deployed

on cloud platforms, such applications navigate a multilayered architecture: from hypervisors and virtual machines to container orchestration, which manages resources, scaling, and health. Each layer introduces complexities affecting data transfer, latency, and throughput.

Understanding the complexities of cloud and containerized environments is crucial for optimizing application performance, necessitating a strategic balance among software capabilities, hardware resources, and the intermediary stack. This pursuit of optimization is further shaped by global technology standards and regulations, emphasizing the importance of security, data protection, and energy efficiency. Such standards drive compliance and spur hardware-level innovations, therby enhancing virtualization stacks and overall system efficiency.

As we transition into this section, our focus expands to the symbiotic relationship between hardware and software, highlighting how hardware intricacies, from data processing mechanisms to memory management strategies, influence the performance of applications. This sets the stage for a comprehensive exploration in the subsequent sections.

Decoding Hardware–Software Dynamics

To maximize the performance of an application, it's essential to understand the complex interplay between its code and the underlying hardware. While software provides the user interface and functionality, the hardware plays a crucial role in determining operational speed. This critical relationship, if not properly aligned, can lead to potential performance bottlenecks.

The characteristics of the hardware, such as CPU speed, memory availability, and disk I/O, directly influence how quickly and effectively an application can process tasks. It's imperative for developers and system architects to be aware of the hardware's capabilities and limitations. For example, an application optimized for multi-core processors might not perform as efficiently on a single-core setup. Additionally, the intricacies of micro-architecture, diverse subsystems, and the variables introduced by cloud environments can further complicate the scaling process.

Diving deeper into hardware–software interactions, we find complexities, especially in areas like hardware architecture and thread behavior. For instance, in an ideal world, multiple threads would seamlessly access a shared memory structure, working in harmony without any conflicts. Such a scenario, depicted in Figure 5.4 and inspired by Maranget et al.'s *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*,[4] while desirable, is often more theoretical than practical.

[4] www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

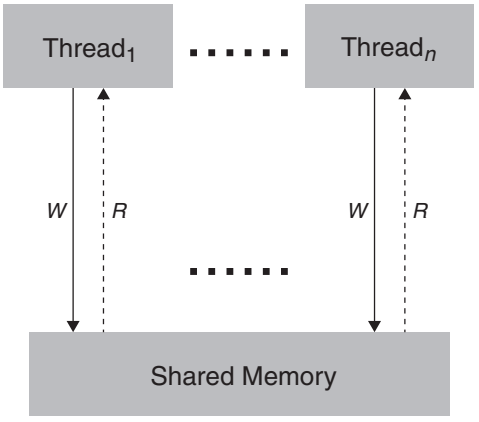

Figure 5.4 Threads 1 to *n* accessing a Shared Memory Structure Without Hindrance (Inspired by Luc Maranget, Susmit Sarkar, and Peter Sewell's *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*)

However, the real world of computing is riddled with scaling and concurrency challenges. Shared memory management often necessitates mechanisms like locks to ensure data validity. These mechanisms, while essential, can introduce complications. Drawing from Doug Lea's *Concurrent Programming in Java: Design Principles and Patterns*,[5] it's not uncommon for threads to vie for locks when accessing objects (Figure 5.5). This competition can lead to scenarios in which one thread inadvertently blocks others, resulting in execution delays and potential performance degradation.

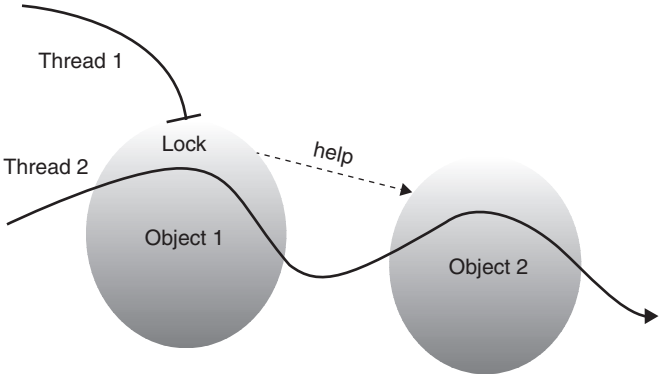

Figure 5.5 Thread 2 Locks Object 1 and Moves to the Helper in Object 2 (Inspired by Doug Lea's *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed.)

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Boston, MA: Addison-Wesley Professional, 1999.

Such disparities between the idealized vision and the practical reality emphasize the intricate interplay and inherent challenges in hardware–software interactions. Threads, shared memory structures, and the mechanisms that govern their interactions (lock or lock-free data structures and algorithms) are a testament to this complexity. It's this divergence that underscores the need for a deep understanding of how hardware influences application performance.

Performance Symphony: Languages, Processors, and Memory Models

At its best, performance is a harmonious composition of programming languages employed, the capabilities of the processors, and the memory models they adhere to. This symphony of performance is what we aim to unravel in this section.

Concurrency: The Core Mechanism

Concurrency refers to the execution of multiple tasks or processes simultaneously. It's a fundamental concept in modern computing, allowing for improved efficiency and responsiveness. From an application standpoint, understanding concurrency is vital for designing systems that can handle multiple users or tasks at once. For web applications, this could mean serving multiple users without delays. Misunderstanding concurrency can lead to issues like deadlocks or race conditions.

The Symbiotic Relationship: Hardware and Software Interplay

- **Beyond hardware:** While powerful hardware can boost performance, the true potential is unlocked when the software is optimized for that hardware. This optimization is influenced by the choice of programming language, its compatibility with the processor, and adherence to a specific memory model.
- **Languages and processors:** Different programming languages have varying levels of abstraction. Whereas some offer a closer interaction with the hardware, others prioritize developer convenience. The challenge lies in ensuring that the code is optimized across this spectrum, making it performant on intended processor architecture.

Memory Models: The Silent Catalysts

- **Balancing consistency and performance:** Memory models define how threads perceive memory operations. They strike a delicate balance between ensuring all threads have a coherent view of memory while allowing for certain reorderings for performance gains.
- **Language specifics:** Different languages come with their own memory models. For instance, Java's memory model emphasizes a specific order of operations to maintain consistency across threads, even if it means sacrificing some performance. By comparison, C++11 and newer versions offer a more fine-grained control over memory ordering than Java.

Enhancing Performance: Optimizing the Harmony

Hardware-Aware Programming

To truly harness the power of the hardware, we must look beyond the language's abstraction. This involves understanding cache hierarchies, optimizing data structures for cache-friendly design, and being aware of the processor's capabilities. Here's how this plays out across different levels of caching:

- **Optimizing for CPU cache efficiency:** Align data structures with cache lines to minimize cache misses and speed up access. Important strategies include managing "true sharing"—minimizing performance impacts by ensuring shared data on the same cache line is accessed efficiently—and avoiding "false sharing," where unrelated data on the same line leads to invalidations. Taking advantage of hardware prefetching, which preloads data based on anticipated use, improves spatial and temporal locality.
- **Application-level caching strategies:** Implementing caching at the software/application level, demands careful consideration of hardware scalability. The available RAM, for instance, dictates the volume of data that can be retained in memory. Decisions regarding what to cache and its retention duration hinge on a nuanced understanding of hardware capabilities, guided by principles of spatial and temporal locality.
- **Database and web caching:** Databases use memory for caching queries and results, linking performance to hardware resources. Similarly, web caching through content delivery networks (CDNs) involves geographically distributed networked servers to cache content closer to users, reducing latency. Software Defined Networks (SDN) can optimize hardware use in caching strategies for improved content delivery efficiency.

Mastering Memory Models

A deep understanding of a language's memory model can prevent concurrency issues and unlock performance optimizations. Memory models dictate the rules governing interactions between memory and multiple threads of execution. A nuanced understanding of these models can significantly mitigate concurrency-related issues, such as visibility problems and race conditions, while also unlocking avenues for performance optimization.

- **Java's memory model:** In Java, grasping the intricacies of the *happens-before* relationship is essential. This concept ensures memory visibility and ordering of operations across different threads. By mastering these relationships, developers can write scalable thread-safe code without excessive synchronization. Understanding how to leverage volatile variables, atomic operations, and synchronized blocks effectively can lead to significant performance improvements, especially in multithreaded environments.
- **Implications for software design:** Knowledge of memory models influences decisions on data sharing between threads, use of locks, and implementation of non-blocking algorithms. It enables a more strategic approach to synchronization, helping developers choose the most appropriate technique for each scenario. For instance, in low-latency systems, avoiding heavy locks and using lock-free data structures can be critical.

Having explored the nuances of hardware and memory models that can help us craft highly efficient and resilient concurrent software, our next step is to delve into the intricacies of memory model specifics.

Memory Models: Deciphering Thread Dynamics and Performance Impacts

Memory models outline the rules governing thread interactions through shared memory and dictate how values propagate between threads. These rules have a significant influence on the system's performance. In an ideal computing world, we would envision the memory model as a sequentially consistent shared memory system. Such a system would ensure the operations are executed in the exact sequence as they appear in the original program order, known as the single, global execution order, resulting in a sequentially consistent machine.

For example, consider a scenario where the program order dictated that Thread 2 would always get the lock on Object 1 first, effectively blocking Thread 1 every time as it moves to the helper in Object 2. This can be visualized with the aid of two complementary diagrammatic representations: a sequence diagram and a Gantt chart.

The sequence diagram shown in Figure 5.6 provides a detailed view of the interactions between threads and objects. It shows Thread 1 initiating its process with a pre-lock operation, immediately followed by Thread 2 acquiring a lock on Object 1. Thread 2's subsequent work on Object 2 and the eventual release of the lock on Object 1 are depicted in a clear, sequential manner. Concurrently, Thread 1 enters a waiting state, demonstrating its dependency on the lock held by Thread 2. Once Thread 2 releases the lock, Thread 1 acquires it, proceeds with its operation on Object 2, and then releases the lock, completing its sequence of operations.

Complementing the sequence diagram, the Gantt chart, shown in Figure 5.7, illustrates the global execution order over a timeline. It presents the start and end points of each thread's interaction with the objects, capturing the essence of sequential consistency. The Gantt chart marks the waiting period of Thread 1, which overlaps with Thread 2's exclusive access period, and then shows the continuation of Thread 1's actions after the lock's release.

By joining the global execution timeline with the program order timeline, we can observe a straightforward, sequential flow of events. This visualization aligns with the principles of a sequentially consistent machine, where despite the concurrent nature of thread operations, the system behaves as if there is a single, global sequence of well-ordered events. The diagrams thus serve to enhance our understanding of sequential consistency.

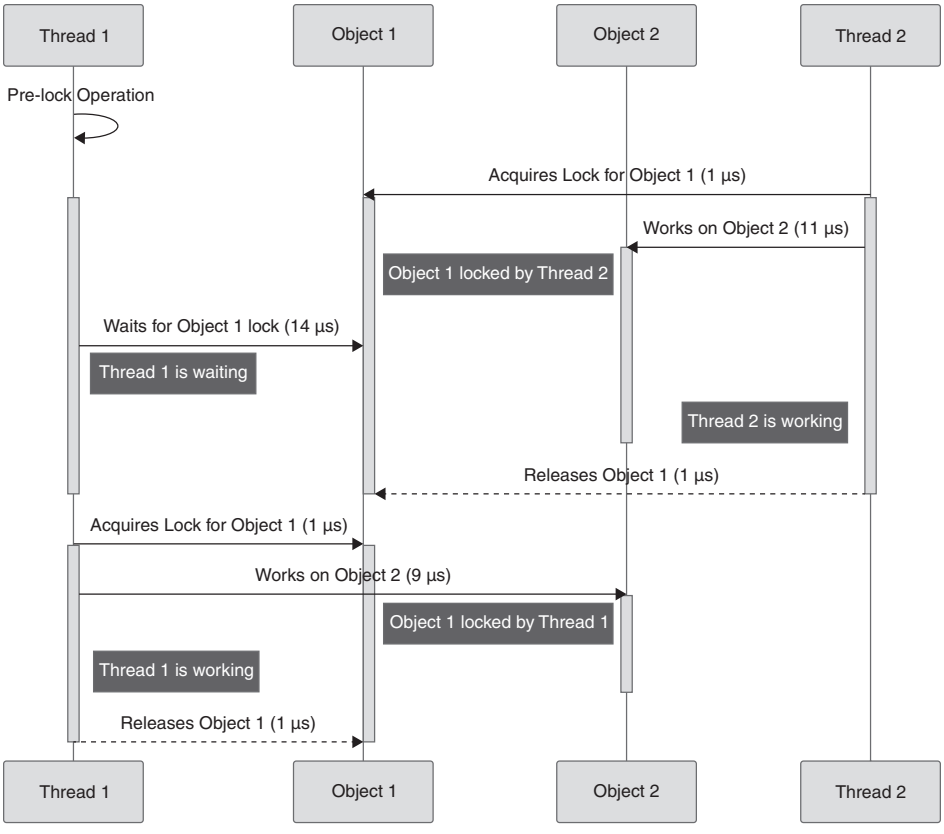

Figure 5.6 Sequence of Thread Operation in a Sequentially Consistent Shared Memory Model

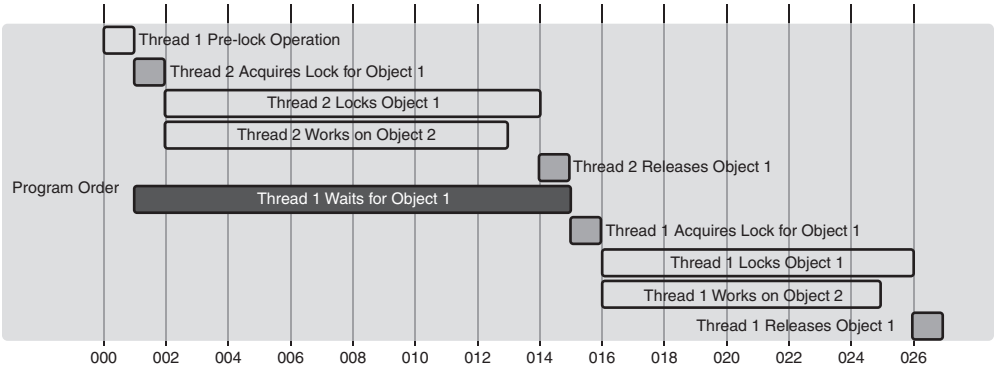

Figure 5.7 Global Execution Timeline of Thread Interactions

While Figures 5.6 and 5.7 and the discussion outline an idealized, sequential consistent system, real-world contemporary computing systems often follow more relaxed memory models such as total store order (TSO), partial store order (PSO), or release consistency. These models, when combined with various hardware optimization strategies such as store/write buffering and out-of-order execution, allow for performance gains. However, they also introduce the possibility of observing operations out of their programmed order, potentially leading to inconsistencies.

Consider the concept of store buffering as an example, where threads may observe their own writes before they become visible to other threads, which can disrupt the perceived order of operations. To illustrate, let's explore this with a store buffering example with two threads (or processes), `p0` and `p1`. Initially, we have two shared variables `X` and `Y` set to `0` in memory. Additionally, we have `foo` and `bar`, which are local variables stored in registers of threads `p0` and `p1`, respectively:

```
p0                  p1
a: X = 1;           c: Y = 1;
b: foo = Y;         d: bar = X;
```

In an ideal, sequentially consistent environment, the values for `foo` and `bar` are determined by combined order of operations that could include sequences like `{a,b,c,d}`, `{c,d,a,b}`, `{a,c,b,d}`, `{c,a,b,d}`, and so on. In such a scenario, it's impossible for both `foo` and `bar` to be zero, as the operations within each thread are guaranteed to be observed in the order they were issued.

Yet, with store buffering, threads might perceive their own writes before others. This can lead to situations in which both `foo` and `bar` are zero, because `p0`'s write to `X` (`a: X = 1`) might be recognized by `p0` before `p1` notices it, and similarly for `p1`'s write to `Y` (`c: Y = 1`).

Although such behaviors can lead to inconsistencies that deviate from the expected sequential order, these optimizations are fundamental for enhancing system performance, paving the way for faster processing times and greater efficiency. Additionally, cache coherence protocols like MESI and MOESI[6] play a crucial role in maintaining consistency across processor caches, further complicating the memory model landscape.

In conclusion, our exploration of memory consistency models has laid the groundwork for understanding the intricacies of concurrent operations within the JVM. For those of us working with Java and JVM performance tuning, it is essential to get a good handle on the practical realities and the underlying principles of these models because they influence the smooth and efficient execution of our concurrent programs.

As we move forward, we will continue to explore the world of hardware. In particular, we'll discuss multiprocessor threads or cores with tiered memory structures alongside the Java Memory Model's (JMM) constructs. We will cover the use of volatile variables - a key JMM feature for crafting thread-safe applications - and explore how concepts like memory barriers and fences contribute to visibility and ordering in concurrent Java environments.

Furthermore, the concept of Non-Uniform Memory Access (NUMA) becomes increasingly relevant as we consider the performance of Java applications on systems with variable memory

[6] https://redis.com/glossary/cache-coherence/

access times. Understanding NUMA's implications is critical, as it affects garbage collection, thread scheduling, and memory allocation strategies within the JVM, influencing the overall performance of Java applications.

Concurrent Hardware: Navigating the Labyrinth

Within the landscape of concurrent hardware systems, we must navigate the complex layers of multiple-processor cores, each of which has its own hierarchical memory structure.

Simultaneous Multithreading and Core Utilization

Multiprocessor cores may implement simultaneous multithreading (SMT), a technique designed to improve the efficiency of CPUs by allowing multiple hardware threads to execute simultaneously on a single core. This concurrency within a single core is designed to better utilize CPU resources, as idle CPU cores can be used by another hardware thread while one thread waits for I/O operations or other long-latency instructions.

Contrasting this with the utilization of full cores, where each thread is run on a separate physical core with dedicated computational resources, we encounter a different set of performance considerations for the JVM. Full cores afford each thread complete command over a core's resources, reducing contention and potentially heightening performance for compute-intensive tasks. In such an environment, JVM's garbage collection and thread scheduling can operate with the assurance of unimpeded access to these resources, tailoring strategies that maximize the efficiency of each core.

However, in the realms where hyper-threading (SMT) is employed, the JVM is presented with a more intricate scenario. It must recognize that threads may not have exclusive access to all the resources of a core. This shared environment can elevate throughput for multithreaded applications but may also introduce a higher likelihood of resource contention. The JVM's thread scheduler and GC must adapt to this reality, balancing workload distribution and GC processes to accommodate the subtleties of SMT.

The choice between leveraging full cores or hyper-threading aligns closely with the nature of the Java application in question. For I/O-bound or multithreaded workloads that can capitalize on idle CPU cycles, hyper-threading may offer a throughput advantage. Conversely, for CPU-bound processes that demand continuous and intensive computation, full cores might provide the best performance outcomes.

As we delve deeper into the implications of these hardware characteristics for JVM performance, we must recognize the importance of aligning our software strategies with the underlying hardware capabilities. Whether optimizing for the dedicated resources of full cores or the concurrent execution environment of SMT, the ultimate goal remains the same: to enhance the performance and responsiveness of Java applications within these concurrent hardware systems.

Advanced Processor Techniques and Memory Hierarchy

The complexity of modern processors extends beyond the realm of SMT and full core utilization. Within these advanced systems, cores have their own unique cache hierarchies to

maintain cache coherence across multiple processing units. Each core typically possesses a Level 1 instruction cache (L1I$), a Level 1 data cache (L1D$), and a private Level 2 cache (L2). Cache coherence is the mechanism that ensures all cores have a consistent view of memory. In essence, when one core updates a value in its cache, other cores are made aware of this change, preventing them from using outdated or inconsistent data.

Included in this setup is a communal last-level cache (LLC) that is typically accessible by all cores, facilitating efficient data sharing among them. Additionally, some high-performance systems incorporate a system-level cache (SLC) alongside or in place of the LLC to further optimize data access times or to serve specific computational demands.

To maximize performance, these processors employ advanced techniques. For example, out-of-order execution allows the processor to execute instructions as soon as the data becomes available, rather than strictly adhering to the original program order. This maximizes resource utilization, improving processor efficiency.

Moreover, as discussed earlier, the processors employ load and store buffers to manage data being transferred to and from memory. These buffers temporarily hold data for memory operations, such as loads and stores, allowing the CPU to continue executing other instructions without waiting for the memory operations to complete. This mechanism smooths out the flow of data between the fast CPU registers and the relatively slower memory, effectively bridging the speed gap and optimizing the processor's performance.

At the heart of the memory hierarchy is a memory controller that oversees the double data rate (DDR) memory banks within the system, orchestrating the data flow to and from physical memory. Figure 5.8 illustrates a comprehensive hardware configuration containing all of the aforementioned components.

Memory consistency, or the manner in which one processor's memory changes are perceived by others, varies across hardware architectures. At one end of the spectrum are architectures with strong memory consistency models like x86-64 and SPARC, which utilize TSO.[7] Conversely, the Arm v8-A architecture has a more relaxed model that allows for more flexible reordering of memory operations. This flexibility can lead to higher performance but requires programmers to be more diligent in using synchronization primitives to avoid unexpected behavior in multi-threaded applications.[8]

To illustrate the challenges, consider the store buffer example discussed earlier with two variables `X` and `Y`, both initially set to `0`. In TSO architectures like x86-64, the utilization of store buffers could lead to both `foo` and `bar` being `0` if thread `P1` reads `Y` before `P2`'s write to `Y` becomes visible, and simultaneously `P2` reads `X` before `P1`'s write to `X` is seen by `P2`. In Arm v8, the hardware could also reorder the operations, potentially leading to even more unexpected outcomes beyond those permissible in TSO.

[7] https://docs.oracle.com/cd/E26502_01/html/E29051/hwovr-15.html

[8] https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/synchronization-overview-and-case-study-on-arm-architecture-563085493

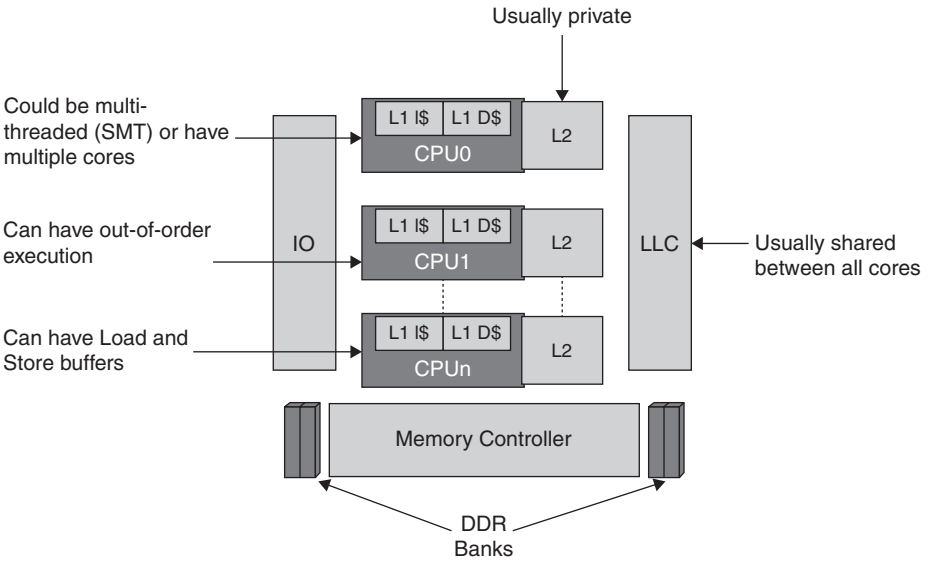

Figure 5.8 A Modern Hardware Configuration

In navigating the labyrinth of advanced processor technologies and memory hierarchies, we have actually uncovered only half the story. The true efficacy of these hardware optimizations is realized when they are effectively synchronized with the software that runs on them. As developers and system architects, our challenge is ensure that our software not only harnesses but also complements these sophisticated hardware capabilities.

To this end, software mechanisms like barriers and fences can both complement and harness these hardware features. These instructions enforce ordering constraints and ensure that operations within a multithreaded environment are executed in a manner that respects the hardware's memory model. This understanding is pivotal for Java performance engineering because it enables us to craft software strategies that optimize the performance of Java applications in complex computing environments. Building on our understanding of Java's memory model and concurrency challenges, let's take a closer look at these software strategies, exploring how they work in tandem with hardware to optimize the efficiency and reliability of Java-based systems.

Order Mechanisms in Concurrent Computing: Barriers, Fences, and Volatiles

In the complex world of concurrent computing, barriers, fences, and volatiles serve as crucial tools to preserve order and ensure data consistency. These mechanisms function by preventing certain types of reordering that could potentially lead to unintended consequences or unexpected outcomes.

Barriers

A barrier is a type of synchronization mechanism used in concurrent programming to block certain kinds of reordering. Essentially, a barrier acts like a checkpoint or synchronization point in your program. When a thread hits a barrier, it cannot pass until all other threads have also reached this barrier. In the context of memory operations, a memory barrier prevents the reordering of read and write operations (loads and stores) across the barrier. This helps in maintaining the consistency of shared data in a multithreaded environment.

Fences

Whereas barriers provide a broad synchronization point, fences are more tailored to memory operations. They enforce ordering constraints, ensuring that certain operations complete before others commence. For instance, a store-load fence ensures that all store operations (writes) that appear before the fence in the program order are visible to the other threads before any load operations (reads) that appear after the fence.

Volatiles

In languages like Java, the `volatile` keyword provides a way to ensure visibility and ordering of variables across threads. With a volatile variable, a write to the variable is visible to all threads that subsequently read from the variable, providing a *happens-before* guarantee. Nonvolatile variables, in contrast, don't have the benefit of the interaction guaranteed by the `volatile` keyword. Hence, the compiler can use a cached value of the nonvolatile variable.

Atomicity in Depth: Java Memory Model and *Happens-Before* Relationship

Atomicity ensures that operations either fully execute or don't execute at all, thereby preserving data integrity. The JMM is instrumental in understanding this concept in a multithreaded context. It provides visibility and ordering guarantees, ensuring atomicity and establishing the *happens-before* relationship for predictable execution. When one action *happens-before* another, the first action is not only visible to the second action but also ordered before it. This relationship ensures that Java programs executed in a multithreaded environment behave predictably and consistently.

Let's consider a practical example that illustrates these concepts in action. The `AtomicLong` class in Java is a prime example of how atomic operations are implemented and utilized in a multithreaded context to maintain data integrity and predictability. `AtomicLong` ensures atomic operations on `long` values through low-level synchronization techniques, acting as single, indivisible units of work. This characteristic protects them from the potential disruption or interference of concurrent operations. Such atomic operations form the bedrock of synchronization, ensuring system-wide coherence even in the face of intense multithreading.

```
AtomicLong counter = new AtomicLong();

void incrementCounter() {
```

```
    counter.incrementAndGet();
}
```

In this code snippet, the `incrementAndGet()` method in the `AtomicLong` class demonstrates atomic operation. It combines the increment (`counter++`) and retrieval (`get`) of the counter value into a single operation. This atomicity is crucial in a multithreaded environment because it prevents other threads from observing or interfering with the `counter` in an inconsistent state during the operation.

The `AtomicLong` class harnesses a low-level technique to ensure this order. It uses compare-and-set loops, rooted in CPU instructions, that effectively act as memory barriers. This atomicity, guaranteed by barriers, is an essential aspect of concurrent programming. As we progress, we'll dig deeper into how these concepts bolster performance engineering and optimization.

Let's consider another practical example to illustrate these concepts, this time involving a hypothetical `DriverLicense` class. This example demonstrates how the `volatile` keyword is used in Java to ensure visibility and ordering of variables across threads:

```
class DriverLicense {
    private volatile boolean validLicense = false;

    void renewLicense() {
        this.validLicense = true;
    }

    boolean isLicenseValid() {
        return this.validLicense;
    }
}
```

In this example, the `DriverLicense` class serves as a model for managing a driver's license status. When the license is renewed (by calling the `renewLicense()` method), the `validLicense` field is set to `true`. The current status can be verified with `isLicenseValid()` method.

The `validLicense` field is declared as `volatile`, which guarantees that once the license is renewed in one thread (for example, after an online renewal), the new status of the license will immediately be visible to all other threads (for example, police officers checking the license status). Without the `volatile` keyword, there might be a delay before other threads could see the updated license status due to various caching or reordering optimizations done by the JVM or the system.

In the context of real-world applications, misunderstanding or improperly implementing these mechanisms can lead to severe consequences. For instance, an e-commerce platform that doesn't handle concurrent transactions properly might double-charge a customer or fail to update inventory correctly. For developers, understanding these mechanisms is essential when designing applications that rely on multithreading. Their proper use can ensure that tasks are completed in the correct order, preventing potential data inconsistencies or errors.

As we advance, our exploration will examine runtime performance through effective thread management and synchronization techniques. Specifically, Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond" will not only introduce advanced synchronization and locking mechanisms but also guide us through the concurrency utilities. By leveraging the strengths of these concurrency frameworks, alongside a foundational understanding of the underlying hardware behavior, we equip ourselves with a comprehensive toolkit for mastering performance optimization in concurrent systems.

Concurrent Memory Access and Coherency in Multiprocessor Systems

In our exploration of memory models and their profound influence on thread dynamics, we've unearthed the intricate challenges that arise when dealing with memory access in multiprocessor systems. These challenges are magnified when multiple cores or threads concurrently access and modify shared data. In such scenarios, it is very important to ensure coherency and efficient memory access.

In the realm of multiprocessor systems, threads often interact with different memory controllers. Although concurrent memory access is a hallmark of these systems, built-in coherency mechanisms ensure that data inconsistencies are kept at bay. For instance, even if one core updates a memory location, coherency protocols will ensure that all cores have a consistent view of the data. This eliminates scenarios where a core might access outdated values. The presence of such robust mechanisms highlights the sophisticated design of modern systems, which prioritize both performance and data consistency.

In multiprocessor systems, ensuring data consistency across threads is a prime concern. Tools like barriers, fences, and volatiles have been foundational in guaranteeing that operations adhere to a specific sequence, maintaining data consistency across threads. These mechanisms are essential in ensuring that memory operations are executed in the correct order, preventing potential *data races* and inconsistencies.

> **NOTE** Data races occur when two or more threads access shared data simultaneously and at least one access is a write operation.

However, with the rise of systems featuring multiple processors and diverse memory banks, another layer of complexity is added—namely, the physical distance and access times between processors and memory. This is where Non-Uniform Memory Access (NUMA) architectures come into the picture. NUMA optimizes memory access based on the system's physical layout. Together, these mechanisms and architectures work in tandem to ensure both data consistency and efficient memory access in modern multiprocessor systems.

NUMA Deep Dive: My Experiences at AMD, Sun Microsystems, and Arm

Building on our understanding of concurrent hardware and software mechanisms, it's time to journey further into the world of NUMA, a journey informed by my firsthand experiences at AMD, Sun Microsystems, and Arm. NUMA is a crucial component in multiprocessing environments, and understanding its nuances can significantly enhance system performance.

Within a NUMA system, the memory controller isn't a mere facilitator—it's the nerve center. At AMD, working on the groundbreaking Opteron processor, I observed how the controller's role was a pivotal function. Such a NUMA-aware, intelligent entity capable of judiciously allocating memory based on the architecture's nuances and the individual processors' needs optimized memory allocation based on proximity to the processors, minimizing cross-node traffic, reducing latency, and enhancing system performance.

Consider this: When a processor requests data, the memory controller decides where the data is stored. If it's local, the controller enables swift, direct access. If not, the controller must engage the interconnection network, thus introducing latency. In my work at AMD, the role of a NUMA-aware memory controller was to strike an optimal balance between these decisions to maximize performance.

Buffers in NUMA systems serve as intermediaries, temporarily storing data during transfers, helping to balance memory access across nodes. This balancing act prevents a single node from being inundated with an excessive number of requests, ensuring a smooth operation—a principle we used to optimize the scheduler for Sun Microsystems' high-performance V40z servers and SPARC T4s.

Attention must also be directed to the transport system—the artery of a NUMA architecture. It enables data mobility across the system. A well-optimized transport system minimizes latency, thereby contributing to the system's overall performance—a key consideration, reinforced during my tenure at Arm.

Mastering NUMA isn't a superficial exercise: It's about understanding the interplay of components such as the memory controllers, buffers, and the transport system within this intricate architecture. This deep understanding has been instrumental in my efforts to optimize performance in concurrent systems.

Figure 5.9 shows a simplified NUMA architecture with buffers and interconnects.

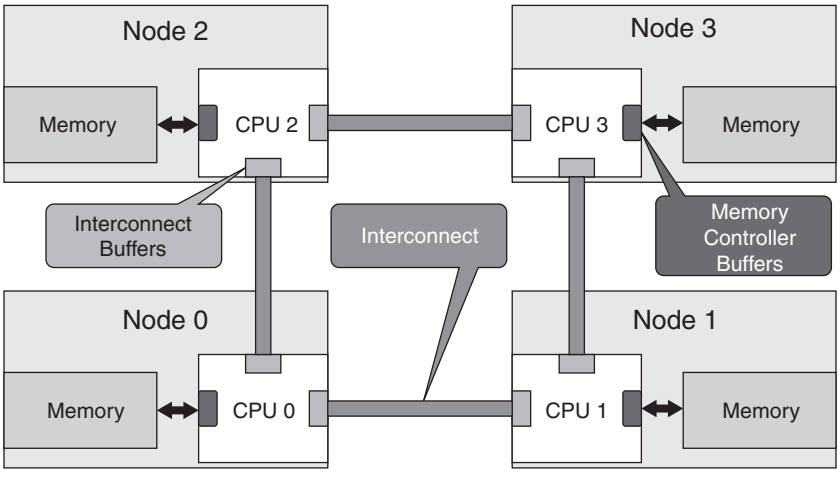

Figure 5.9 NUMA Nodes in Modern Systems

Venturing further into the NUMA landscape, we encounter various critical traffic patterns, including cross-traffic, local traffic, and interleaved memory traffic patterns. These patterns significantly influence memory access latency and, consequently, data processing speed (Figure 5.10).

Cross-traffic refers to instances where a processor accesses data in another processor's memory bank. This cross-node movement introduces latency but is an inevitable reality of multiprocessing systems. A key consideration in such cases is efficient handling through intelligent memory allocation, facilitated by the NUMA-aware memory controller and observing paths with the least latency, without crowding the buffers.

Local traffic, by contrast, occurs when a processor retrieves data from its local memory bank—a process faster than cross-node traffic due to the elimination of the interconnect network's demands. Optimizing a system to increase local traffic while minimizing cross-traffic latencies is a cornerstone of efficient NUMA systems.

Interleaved memory adds another layer of complexity to NUMA systems. Here, successive memory addresses scatter across various memory banks. This scattering balances the load across nodes, reducing bottlenecks and enhancing performance. However, it can inadvertently increase cross-node traffic—a possibility that calls for careful management of the memory controllers.

NUMA architecture addresses the scalability limitations inherent in shared memory models, wherein all processors share a single memory space. As the number of processors increases, the bandwidth to the shared memory becomes a bottleneck, limiting the scalability of the system. However, programming for NUMA systems can be challenging, as developers need to consider memory locality when designing their algorithms and data structures. To assist developers modern operating systems offer NUMA APIs for allocating memory on specific nodes. Additionally, memory management systems, such as databases and GCs, are becoming increasingly NUMA-aware. This awareness is crucial for maximizing system performance and was a key focus in my work at AMD, Sun Microsystems, and Arm.

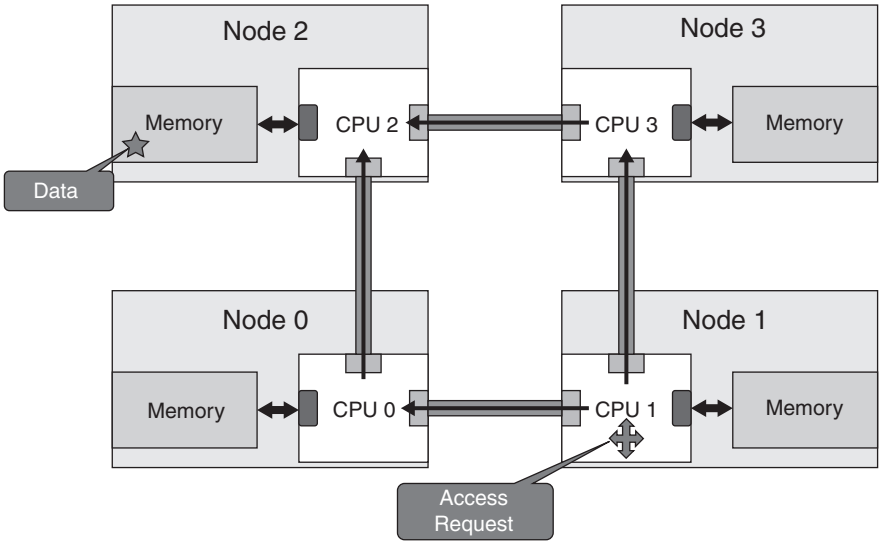

Figure 5.10 An Illustration of NUMA Access

Other related concepts include Cache Coherent Non-Uniform Memory Access (ccNUMA), in which a system maintains a consistent image of memory across all processors; by comparison, NUMA refers to the overarching design philosophy of building systems with non-uniform memory. Understanding these concepts, in addition to barriers, fences, and volatiles, is critical to mastering performance optimization in concurrent systems.

Advancements in NUMA: Fabric Architectures in Modern Processors

The NUMA architecture has undergone many advancements with the evolution of processor designs, which are particularly noticeable in high-end server processors. These innovations are readily evident in the work done by industry giants like Intel, AMD, and Arm, each of which has made unique contributions that are particularly relevant in server environments and cloud computing scenarios.

- **Intel's mesh fabric:** The Intel Xeon Scalable processors have embraced a mesh architecture, which replaces the company's traditional ring interconnect.[9] This mesh topology effectively connects cores, caches, memory, and I/O components in a more direct manner. This allows for higher bandwidth and lower latency, both of which are crucial for data center and high-performance computing applications. This shift significantly bolsters data-processing capabilities and system responsiveness in complex computational environments.
- **AMD's infinity fabric:** AMD's EPYC processors feature the Infinity Fabric architecture.[10] This technology interconnects multiple processor dies within the same package, enabling efficient data transfer and scalability. It's particularly beneficial in multiple-die CPUs, addressing challenges related to exceptional scaling and performance in multi-core environments. AMD's approach dramatically improves data handling in large-scale servers, which is critical to meet today's growing computational demands.
- **Arm's NUMA advances:** Arm has also made strides in NUMA technology, particularly in terms of its server-grade CPUs. Arm architectures like the Neoverse N1 platform[11] embody a scalable architecture for high-throughput, parallel-processing tasks with an emphasis on an energy-efficient NUMA design. The intelligent interconnect architecture optimizes memory accesses across different cores, balancing performance with power efficiency, and thereby improving performance per watts for densely populated servers.

From Pioneering Foundations to Modern Innovations: NUMA, JVM, and Beyond

Around 20 years ago, during my tenure at AMD, our team embarked on a groundbreaking journey. We ventured beyond the conventional, seeking to bridge the intricate world of NUMA with the dynamic realm of JVM performance. This early collaboration culminated in the inception of the NUMA-aware GC, a significant innovation to strategically optimize memory management in such a multi-node system. This was a testament to the foresight and creativity

[9] www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html

[10] www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf

[11] www.arm.com/-/media/global/solutions/infrastructure/arm-neoverse-n1-platform.pdf

of that era. The insights we garnered then weren't just momentary flashes of brilliance; they laid a robust foundation for the JVM landscape we're familiar with today. Today, we're building upon that legacy, enhancing various GCs with the wisdom and insights from our foundational work. In Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK," you'll witness how these early innovations continue to shape and influence modern JVM performance.

Bridging Theory and Practice: Concurrency, Libraries, and Advanced Tooling

As we shift our focus to practical application of these foundational theories, the insights gained from understanding the architectural nuances and concurrency paradigms will help us build tailor-made frameworks and libraries. These can take advantage of the SMT or per-core scaling and prime the software cache to complement hardware cache/memory hierarchies. Armed with this knowledge, we should be able to harmonize our systems with the underlying architectural model, using it to our advantage with respect to thread pools or to harness concurrent, lock-free algorithms. This approach, a vital aspect of cloud computing, is a perfect example of what it means to be cloud-native in application and libraries development.

The JVM ecosystem comes with advanced tools for monitoring and profiling, which can be instrumental in translating these theoretical concepts into actionable insights. Tools like Java Mission Control and VisualVM allow developers to peek under the hood of the JVM, offering a detailed understanding of how applications interact with the JVM and underlying hardware. These insights are crucial for identifying bottlenecks and optimizing performance.

Performance Engineering Methodology: A Dynamic and Detailed Approach

When we examine the previous sections through the lens of application development, two common threads emerge: optimization and efficiency. While these concepts might initially seem abstract or overly technical, they are central to determining how well an application performs, its scalability, and its reliability. Developers focused solely on the application layer might inadvertently underestimate the significance of these foundational concepts. However, a holistic understanding of both the application itself and the underlying system infrastructure can ensure that the resulting software is robust, efficient, and capable of meeting user demands.

Performance engineering—a multidimensional, cyclical discipline—necessitates a deep comprehension of not just the application, JVM, and OS, but also the underlying hardware. In today's diverse technological landscape, this domain extends to include modern deployment environments such as containers and virtual machines managed by hypervisors. These elements, which are integral to cloud computing and virtualized infrastructures, add layers of complexity and opportunity to the performance engineering process.

Within this expanded domain, we can employ diverse methodologies, such as top-down and bottom-up approaches, to uncover, diagnose, and address performance issues entrenched in systems, whether they are running on traditional setups or modern, distributed architectures. Alongside these approaches, the technique of experimental design—although not a traditional methodology—plays a pivotal role, offering a systematic procedure for conducting experiments and evaluating system performance from baremetal servers to containerized microservices. These approaches and techniques are universally applicable, ensuring that performance optimization remains achievable regardless of the deployment scenarios.

Experimental Design

Experimental design is a systematic approach used to test hypotheses about system performance. It emphasizes data collection and promotes evidence-based decision making. This method is employed in both top-down and bottom-up methodologies to establish baseline and peak or stress performance measurements and also to precisely evaluate the impact of incremental or specific enhancements. The baseline performance represents the system's capabilities under normal or expected loads, whereas the peak or stress performance highlights its resilience under extreme loads. Experimental design aids in distinguishing variable changes, assessing performance impacts, and gathering crucial monitoring and profiling data.

Inherent to experimental design are the control and treatment conditions. The control condition represents the baseline scenario, often corresponding to the system's performance under a typical load, such as with the current software version. In contrast, the treatment condition introduces experimental variables, such as a different software version or load, to observe their impact. The comparison between the control and the treatment conditions helps gauge the impact of the experimental variables on system performance.

One specific type of experimental design is A/B testing, in which two versions of a product or system are compared to determine which performs better in terms of a specific metric. While all A/B tests are a form of experimental design, not all experimental designs are A/B tests. The broader field of experimental design can involve multiple conditions and is used across various domains, from marketing to medicine.

A crucial aspect of setting up these conditions involves understanding and controlling the state of the system. For instance, when measuring the impact of a new JVM feature, it's essential to ensure that the JVM's state is consistent across all trials. This could include the state of the GC, the state of JVM-managed threads, the interaction with the OS, and more. This consistency ensures reliable results and accurate interpretations.

Bottom-Up Methodology

The bottom-up methodology begins at the granularity of low-level system components and gradually works its way up to provide detailed insights into system performance. Throughout my experiences at AMD, Sun, Oracle, Arm, and now at Microsoft as a JVM system and GC performance engineer, the bottom-up methodology has been a crucial tool in my repertoire. Its utility has been evident in a multitude of scenarios, from introducing novel features at the JVM/runtime level and evaluating the implications of Metaspace, to fine-tuning thresholds

for the G1 GC's mixed collections. As we'll see in the upcoming JMH benchmarking section, the bottom-up approach was also critical in understanding how a new Arm processor's atomic instruction set could influence the system's scalability and performance.

Like experimental design, the bottom-up approach requires a thorough understanding of how state is managed across different levels of the system, extending from hardware to high-level application frameworks:

- **Hardware nuances:** It's essential to comprehend how processor architectures, memory hierarchies, and I/O subsystems impact the JVM behavior and performance. For example, understanding how a processor's core and cache architecture influences garbage collection and thread scheduling in the JVM can lead to more informed decisions in system tuning.
- **JVM's GC:** Developers must recognize how the JVM manages heap memory, particularly how garbage collection strategies like G1 and low-latency collectors like Shenandoah and ZGC are optimized for different hardware, OS, and workload patterns. For instance, ZGC enhances performance by utilizing large memory pages and employing multi-mapping techniques, with tailored optimizations that adapt to the specific memory management characteristics and architectural nuances of different operating systems and hardware platforms.
- **OS threads management:** Developers should explore how the OS manages threads in conjunction with the JVM's multithreading capabilities. This includes the nuances of thread scheduling and the effects of context switching on JVM performance.
- **Containerized environments:**
  - **Resource allocation and limits:** Analyzing container orchestration platforms like Kubernetes for their resource allocation strategies—from managing CPU and memory requests and limits to their interaction with the underlying hardware resources. These factors influence crucial JVM configurations and impact their performance within containers.
  - **Network and I/O operations:** Evaluating the impact of container networking on JVM network-intensive applications and the effects of I/O operations on JVM's file handling.
- **JVM tuning in containers:** We should also consider the specifics of JVM tuning in container environments, such as adjusting heap size with respect to the container's memory limits. In a resource-restricted environment, ensuring that the JVM is properly tuned within the container is crucial.
- **Application frameworks:** Developers must understand how frameworks like Netty, used for high-performance network applications, manage asynchronous tasks and resource utilization. In a bottom-up analysis, we would scrutinize Netty's thread management, buffer allocation strategies, and how these interact with JVM's non-blocking I/O operations and GC behavior.

The efficacy of the bottom-up methodology becomes particularly evident when it's integrated with benchmarks. This combination enables precise measurement of performance and scalability across the entire diverse technology stack—from the hardware and the OS, up through the JVM, container orchestration platforms, and cloud infrastructure, and finally to

the benchmark itself. This comprehensive approach provides an in-depth understanding of the performance landscape, particularly highlighting the crucial interactions of the JVM with the container within the cloud OS, the memory policies, and the hardware specifics. These insights, in turn, are instrumental in tuning the system for optimal performance. Moreover, the bottom-up approach has played an instrumental role in enhancing the out-of-box user experience for JVM and GCs, reinforcing its importance in the realm of performance engineering. We will explore this pragmatic methodology more thoroughly in Chapter 7, when we discuss performance improvements related to contented locks.

In contrast, the top-down methodology, our primary focus in the next sections, is particularly valuable when operating under a definitive statement of work (SoW). A SoW is a carefully crafted document that delineates the responsibilities of a vendor or service provider. It outlines the expected work activities, specific deliverables, and timeline for execution, thereby serving as a guide/roadmap for the performance engineering endeavor. In the context of performance engineering, the SoW provides a detailed layout for achieving the system's quality driven SLAs. This methodology, therefore, enables system-specific optimizations to be aligned coherently with the overarching goals as outlined in the SoW.

Taken together, these methodologies and techniques equip a performance engineer with a comprehensive toolbox, facilitating the effective enhancement of system throughput and the extension of its capabilities.

Top-Down Methodology

The top-down methodology is an overarching approach that begins with a comprehensive understanding of the system's high-level objectives and works downward, leveraging our in-depth understanding of the holistic stack. The method thrives when we have a wide spectrum of knowledge—from the high-level application architecture and user interaction patterns to the finer details of hardware behavior and system configuration.

The approach benefits from its focus on "known-knowns"—the familiar forces at play, including operational workflows, access patterns, data layouts, application pressures, and tunables. These "known-knowns" represent aspects or details about the system that we already understand well. They might include the structure of the system, how its various components interact, and the general behavior of the system under normal conditions. The strength of the top-down methodology lies in its ability to reveal the "known-unknowns"—that is, aspects or details about the system that we know exist, but we don't understand well yet. They might include potential performance bottlenecks, unpredictable system behaviors under certain conditions, or the impact of specific system changes on performance.

> **NOTE** The investigative process embedded in the top-down methodology aims to transform the "known-unknowns" into "known-knowns," thereby providing the knowledge necessary for continual system optimization. While our existing domain knowledge of the subsystems provides a foundational understanding, effectively identifying and addressing various bottlenecks requires more detailed and specific information that goes beyond this initial knowledge to help us achieve our performance and scalability goals.

Applying the top-down methodology demands a firm grasp of how state is managed across different system components, from the higher-level application state down to the finer details of memory management by the JVM and the OS. These factors can significantly influence the system's overall behavior. For instance, when bringing up a new hardware system, the top-down approach starts by defining the performance objectives and requirements of the hardware, followed by investigating the "known-unknowns," such as scalability, and optimal deployment scenarios. Similarly, transitioning from an on-premises setup to a cloud environment begins with the overarching goals of the migration, such as cost-efficiency and low overhead, and then addresses how these objectives impact application deployment, JVM tuning, and resource allocation. Even when you're introducing a new library into your application ecosystem, you should employ the top-down methodology, beginning with the desired outcomes of the integration and then examining the library's interactions with existing components and their effects on application behavior.

Building a Statement of Work

As we transition to constructing a statement of work, we will see how the top-down methodology shapes our approach to optimizing complex systems like large-scale online transactional processing (OLTP) database systems. Generally, such systems, which are characterized by their extensive, distributed data footprints, require careful optimization to ensure efficient performance. Key to this optimization, especially in the case of our specific OLTP system, is the effective cache utilization and multithreading capabilities, which can significantly reduce the system's *call chain traversal time*.

Call chain traversal time is a key metric in OLTP systems, representing the cumulative time required to complete a series of function calls and returns with in a program. This metric is calculated by summing the duration of all individual function calls in the chain. In environments where rapid processing of transactions is essential, minimizing this time can significantly enhance the system's performance.

Creating a SoW for such a complex system involves laying out clear performance and capacity objectives, identifying possible bottlenecks, and specifying the timeline and deliverables. In our OLTP system, it also requires understanding how state is managed within the system, especially when the OLTP system handles a multitude of transactions and needs to maintain consistent stateful interactions. How this state is managed across various components—from the application level to the JVM, OS, and even at the I/O process level—has a significant bearing on the system's performance. The SoW acts as a strategic roadmap, guiding our use of the top-down methodology in investigating and resolving performance issues.

The OLTP system establishes object data relationships for rapid retrievals. The goal is not only to enhance transactional latency but also to balance this with the need to reduce *tail latencies* and increase system utilization during peak loads. Tail latency refers to the delay experienced at the worst percentile of a distribution (for example, the 99th percentile and above). Reducing tail latency means making the system's worst-case performance better, which can greatly improve the overall user experience.

Broadly speaking, OLTP systems often aim to improve responsiveness and efficiency simultaneously. They strive to ensure that each transaction is processed swiftly (enhancing responsiveness) and that the system can handle an increasing number of these transactions, especially during periods of high demand (ensuring efficiency). This dual focus is crucial for the system's overall operational effectiveness, which in turn makes it an essential aspect of the performance engineering process.

Balancing responsiveness with system efficiency is essential. If a system is responsive but not scalable, it might process individual tasks quickly, but it could become overwhelmed when the number of tasks increases. Conversely, a system that is scalable but not performant could handle a large number of tasks, but if each task takes too long to process, the overall system efficiency will still be compromised. Therefore, the ideal system excels in both aspects.

In the context of real-world Java workloads, these considerations are particularly relevant. Java-based systems often deal with high transaction volumes and require careful tuning to ensure optimal performance and capacity. The focus on reducing tail latencies, enhancing transactional latency, and increasing system utilization during peak loads are all critical aspects of managing Java workloads effectively.

Having established the clear performance and scalability goals in our SoW, we now turn to the process that will enable us to meet these objectives. In the next section, we take a detailed look at the performance engineering process, systematically exploring the various facets and subsystems at play.

The Performance Engineering Process: A Top-Down Approach

In this section, we will embark on our exploration of performance engineering with a systematic, top-down approach. This methodology, chosen for its comprehensive coverage of system layers, begins at the highest system level and subsequently drills down into more specific subsystems and components. The cyclic nature of this methodology involves four crucial phases:

1. **Performance monitoring:** Tracking the application's performance metrics over its lifespan to identify potential areas for improvement.
2. **Profiling:** Carrying out targeted profiling for each pattern of interest as identified during monitoring. Patterns may need to be repeated to gauge their impact accurately.
3. **Analysis:** Interpreting the results from the profiling phase and formulating a plan of action. The analysis phase often goes hand in hand with profiling.
4. **Apply tunings:** Applying the tunings suggested by the analysis phase.

The iterative nature of this process ensures that we refine our understanding of the system, continuously improve performance, and meet our scalability goals. Here, it's important to remember that hardware monitoring can play a critical role. Data collected on CPU utilization, memory usage, network I/O, disk I/O, and other metrics can provide valuable insights into the system's performance. These findings can guide the profiling, analysis, and tuning stages, helping us understand whether a performance issue is related to the software or due to hardware limitations.

Building on the Statement of Work: Subsystems Under Investigation

In this section, we apply our top-down performance methodology to each layer of the modern application stack, represented in Figure 5.11. This systematic approach allows us to enhance the call chain traversal times, mitigate tail latencies, and increase system utilization during peak loads.

Our journey through the stack begins with the *Application* layer, where we assess its functional performance and high-level characteristics. We then explore the *Application Ecosystem*, focusing on the libraries and frameworks that support the application's operations. Next, we focus on the *JVM's* execution capabilities and the *JRE*, responsible for providing the necessary libraries and resources for execution. These components could be encapsulated within the *Container*, ensuring that the application runs quickly and reliably from one computing environment to another.

Beneath the container lies the *(Guest) OS*, tailored to the containerized setting. It is followed by the *Hypervisor* along with the *Host OS* for virtualized infrastructures, which manage the containers and provide them access to the physical *Hardware* resources.

We evaluate the overall system performance by leveraging various tools and techniques specific to each layer to determine where potential bottlenecks may lie and look for opportunities to make broad-stroke improvements. Our analysis encompasses all elements, from hardware to software, providing us with a holistic view of the system. Each layer's analysis, guided by the goals outlined in our SoW, provides insights into potential performance bottlenecks and optimization opportunities. By systematically exploring each layer, we aim to enhance the overall performance and scalability of the system, aligning with the top-down methodology's objective to transform our system-level understanding into targeted, actionable insights.

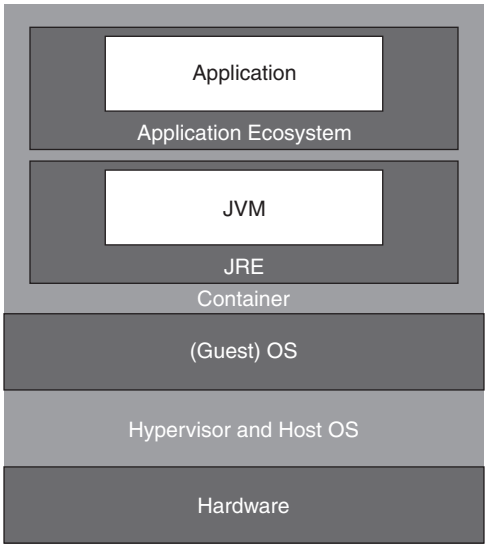

Figure 5.11 A Simplified Illustration of Different Layers of a Typical System Stack

The Application and Its Ecosystem: A System-Wide Outlook

In line with our SoW for the Java-centric OLTP database system, we'll now focus on the application and its ecosystem for a thorough system-wide examination. Our analysis encompasses the following key components:

- **Core database services and functions:** These are the fundamental operations that a database system performs, such as data storage, retrieval, update, and delete operations.
- **Commit log handling:** This refers to the management of commit logs, which record all transactions that modify the data in a database.
- **Data caches:** These hardware or software components store data so future requests for that data can be served faster, which can significantly enhance the system's performance.
- **Connection pools:** These caches of database connections are maintained and optimized for reuse. This strategy promotes efficient utilization of resources, as it mitigates the overhead of establishing a new connection every time database access is required.
- **Middleware/APIs:** Middleware serves as a software bridge between an OS and the applications that run on it, facilitating seamless communications and interactions. In tandem, application programming interfaces (APIs) provide a structured set of rules and protocols for constructing and interfacing with software applications. Collectively, middleware and APIs bolster the system's interoperability, ensuring effective and efficient interaction among its various components.
- **Storage engine:** This is the underlying software component that handles data storage and retrieval in the database system. Optimizing the storage engine can lead to significant performance improvements.
- **Schema:** This is the organization or structure for a database, defined in a formal language supported by the database system. An efficient schema design can improve data retrieval times and overall system performance.

System Infrastructure and Interactions

In a software system, libraries are collections of precompiled code that developers can reuse to perform common tasks. Libraries and their interactions play a crucial role in the performance of a system. Interactions between libraries refer to how these different sets of precompiled code work together to execute more complex tasks. Examining these interactions can provide important insights into system performance, as problems or inefficiencies in these interactions could potentially slow down the system or cause it to behave in unexpected ways.

Now, let's link these libraries and their interactions to the broader context of the operational categories of a database system. Each of the components in the database system contributes to the following:

1. **Data preprocessing tasks:** Operations such as standardization and validation, which are crucial for maintaining data integrity and consistency in OLTP systems. For these tasks, we aim to identify the most resource-intensive transactions and map their paths. We'll also try to understand the libraries involved and how they interact with one another.

2. **Data storage or preparation tasks:** Operations needed to store or prepare data for quick access and modifications for OLTP systems that handle a large number of transactions. Here, we'll identify critical data transfer and data feed paths to understand how data flows within the system.
3. **Data processing tasks:** The actual work patterns, such as sorting, structuring, and indexing, which are directly tied to the system's ability to execute transactions promptly. For these tasks, we'll identify critical data paths and transactions. We'll also pinpoint caches or other optimization techniques used to accelerate these operations.

For each category, we will gather the following information:

- Costly transactions and their paths
- Data transfer and data feed paths
- Libraries and their interactions
- Caches or other optimization techniques to accelerate transactions or data paths

By doing so, we can gain insights into where inefficiencies might lie, and which components might benefit from performance optimization. The categorized operations—data preprocessing tasks, data storage or preparation tasks, and data processing tasks—are integral parts of a database system. Enhancing these areas will help improve transaction latency (performance) and allow the system to handle an increasing number of transactions (scalability).

For example, identifying costly transactions (those that take a long time to process or that lock up other critical operations) and their paths can help streamline data preprocessing, reduce tail latencies, and improve the call chain traversal time. Similarly, optimizing data storage or preparation tasks can lead to quicker retrievals and efficient state management. Identifying critical data transfer paths and optimizing them can significantly increase system utilization during peak loads. Lastly, focusing on data processing tasks and optimizing the use of caches or other techniques can expedite transactions and improve overall system scalability.

With an understanding of the system-wide dynamics, we can now proceed to focus on specific components of the system. Our first stop: the JVM and its runtime environment.

JVM and Runtime Environment

The JVM serves as a bridge between our application and the container, OS, and hardware it runs on—which means that optimizing the JVM can result in significant performance gains. Here, we discuss the configuration options that most directly impact performance, such as JIT compiler optimizations, GC algorithms, and JVM tuning.

We'll start by identifying the JDK version, JVM command line, and GC deployment information. The JVM command line refers to the command-line options that can be used when starting a JVM. These options can control various aspects of JVM behavior, such as setting the maximum heap size or enabling various logging options, as elaborated in Chapter 4, "The Unified Java Virtual Machine Logging Interface." The GC deployment information refers to the configuration and operational details of the GC in use by the JVM, such as the type of GC, its settings, and its performance metrics.

Considering the repetitive and transaction-intensive nature of our OLTP system, it's important to pay attention to JIT compiler optimizations like method inlining and loop optimizations. These optimizations are instrumental in reducing method call overheads and enhancing loop execution efficiency, both of which are critical for processing high-volume transactions swiftly and effectively.

Our primary choice for a GC is the Garbage First Garbage Collector (G1 GC), owing to its proficiency in managing the short-lived transactional data that typically resides in the young generation. This choice aligns with our objective to minimize tail latencies while efficiently handling numerous transactions. In parallel, we should evaluate ZGC and keep a close watch on the developments pertaining to its generational variant, which promises enhanced performance suitable for OLTP scenarios.

Subsequently, we will gather the following data:

- GC logging to help identify GC thread scaling, transient data handling, phases that are not performing optimally, GC locker impact, time to safe points, and application stopped times outside of GC
- Thread and locking statistics to gain insights into thread locking and to get a better understanding of the concurrency behaviors of our system
- Java Native Interface (JNI) pools and buffers to ensure optimal interfacing with native code
- JIT compilation data on inline heuristics and loop optimizations, as they are pivotal for accelerating transaction processing

NOTE We'll gather profiling data separately from the monitoring data due to the potentially intrusive nature of profiling tools like Java Flight Recorder and VisualVM.

Garbage Collectors

When we talk about performance, we inevitably land on a vital topic: optimization of GCs. Three elements are crucial in this process:

- **Adaptive reactions:** The ability of the GC to adjust its behavior based on the current state of the system, such as adapting to higher allocation rates or increased long-lived data pressures.
- **Real-time predictions:** The ability of the GC to make predictions about future memory usage based on current trends and adjust its behavior accordingly. Most latency-geared GCs have this prediction capability.
- **Fast recovery**: The ability of the GC to quickly recover from situations where it is unable to keep up with the rate of memory allocation or promotion, such as by initiating an evacuation failure or a full garbage collection cycle.

Containerized Environments

Containers have become a ubiquitous part of modern application architecture, significantly influencing how applications are deployed and managed. In a containerized environment, JVM ergonomics takes on a new dimension of complexity. The orchestration strategies utilized, particularly in systems like Kubernetes, demand a nuanced approach to JVM resource allocation, aligning it with the performance objectives of our OLTP system. Here, we delve into the following aspects of containerized environments:

- **Container resource allocation and GC optimization:** We build upon the container orchestration strategies to ensure that the JVM's resource allocation is aligned with the OLTP system's performance objectives. This includes manual configurations to override JVM ergonomics that might not be optimal in a container setup.
- **Network-intensive JVM tuning:** Given the network-intensive nature of OLTP systems, we evaluate how container networking impacts JVM performance and optimize network settings accordingly.
- **Optimizing JVM's I/O operations:** We scrutinize the effects of I/O operations, particularly file handling by the JVM, to ensure efficient data access and transfer in our OLTP system.

OS Considerations

In this section, we analyze how the OS can influence the OLTP system's performance. We explore the nuances of process scheduling, memory management, and I/O operations, and discuss how these factors interact with our Java applications, especially in high-volume transaction environments. We'll employ a range of diagnostic tools provided by the OS to gather data at distinct levels and group the information based on the categories identified earlier.

For instance, when working with Linux systems, we have an arsenal of effective tools at our disposal:

- **vmstat:** This utility is instrumental in monitoring CPU and memory utilization, supplying a plethora of critical statistics:
  - **Virtual memory management:** A memory management strategy that abstracts the physical storage resources on a machine, creating an illusion of extensive main memory for the user. This technique significantly enhances the system's efficiency and flexibility by optimally managing and allocating memory resources according to needs.
  - **CPU usage analysis:** The proportion of time the CPU is operational, compared to when it is idle. A high CPU usage rate typically indicates that the CPU is frequently engaged in transaction processing tasks, whereas a low value may suggest underutilization or an idle state, often pointing toward other potential bottlenecks.

- **Sysstat tools**
  - **mpstat:** Specifically designed for thread- and CPU-level monitoring, *mpstat* is adept at gathering information on lock contention, context switches, wait time, steal time, and other vital CPU/thread metrics relevant to high-concurrency OLTP scenarios.
  - **iostat:** Focuses on input/output device performance, providing detailed metrics on disk read/write activities, crucial for diagnosing IO bottlenecks in OLTP systems.

These diagnostic tools not only provide a snapshot of the current state but also guide us in fine-tuning the OS to enhance the performance of our OLTP system. They help us understand the interaction between the OS and our Java applications at a deeper level, particularly in managing resources efficiently under heavy transaction loads.

Host OS and Hypervisor Dynamics

The host OS and hypervisor subsystem is the foundational layer of virtualization. The host OS layer is crucial for managing the physical resources of the machine. In OLTP systems, its efficient management of CPU, memory, and storage directly impacts overall system performance. Monitoring tools at this level help us understand how well the host OS is allocating resources to different VMs and containers.

The way the hypervisor handles CPU scheduling, memory partitioning, and network traffic can significantly affect transaction processing efficiency. Hence, to optimize OLTP performance, we analyze the hypervisor's settings and its interaction with both the host OS and the guest OS. This includes examining how virtual CPUs are allocated, understanding the memory overcommitment strategies, and assessing network virtualization configurations.

Hardware Subsystem

At the lowest level, we consider the actual physical hardware on which our software runs. This examination builds on our foundational knowledge of different CPU architectures and the intricate memory subsystems. We also look at the storage and network components that contribute to the OLTP system's performance. In this context, the concepts of concurrent programming and memory models we previously discussed become even more relevant. By applying effective monitoring techniques, we aim to understand resource constraints or identify opportunities to improve performance. This provides us with a complete picture of the hardware–software interactions in our system.

Thus, a thorough assessment of the hardware subsystem—covering CPU behavior, memory performance, and the efficiency of storage and network infrastructure—is essential. This approach helps by not only pinpointing performance bottlenecks but also illuminating the intricate relationships between the software and hardware layers. Such insights are crucial for implementing targeted optimizations that can significantly elevate the performance of our Java-centric OLTP database system.

To deepen our understanding and facilitate this optimization, we explore some advanced tools that bridge the gap between hardware assessment and actionable data: system profilers and performance monitoring units. These tools play a pivotal role in translating raw hardware metrics into meaningful insights that can guide our optimization strategies.

System Profilers and Performance Monitoring Units

Within the realm of performance engineering, system profilers—such as `perf`, OProfile, Intel's VTune Profiler, and other architecture-geared profilers—provide invaluable insights into hardware's functioning that can help developers optimize both system and application performance. They interact with the hardware subsystem to provide a comprehensive view that is unattainable through standard monitoring tools alone.

A key source of data for these profilers is the performance monitoring units (PMUs) within the CPU. PMUs, including both fixed-function and programmable PMUs, provide a wealth of low-level performance information that can be used to identify bottlenecks and optimize system performance.

System profilers can collect data from PMUs to generate a detailed performance profile. For example, the `perf` command can be used to sample CPU stack traces across the entire system, creating a "flat profile" that summarizes the time spent in each function across the entire system. This kind of performance profile can be used to identify functions that are consuming a disproportionate amount of CPU time. By providing both inclusive and exclusive times for each function, a flat profile offers a detailed look at where the system spends its time.

However, system profilers are not limited to CPU performance; they can also monitor other system components, such as memory, disk I/O, and network I/O. For example, tools like *iostat* and *vmstat* can provide detailed information about disk I/O and memory usage, respectively.

Once we have identified problematic patterns, bottlenecks, or opportunities for improvement through monitoring stats, we can employ these advanced profiling tools for deeper analysis.

Java Application Profilers

Java application profilers like *async-profiler*[12] (a low-overhead sampling profiler) and VisualVM (a visual tool that integrates several command-line JDK tools and provides lightweight profiling capabilities) provide a wealth of profiling data on JVM performance. They can perform different types of profiling, such as CPU profiling, memory profiling, and thread profiling. When used in conjunction with other system profilers, these tools can provide a more complete picture of application performance, from the JVM level to the system level.

By collecting data from PMUs, profilers like *async-profiler* can provide low-level performance information that helps developers identify bottlenecks. They allow developers to get stack and generated code-level details for the workload, enabling them to compare the time spent in the generated code and compiler optimizations for different architectures like Intel and Arm.

To use *async-profiler* effectively, we need *hsdis*,[13] a disassembler plug-in for OpenJDK HotSpot VM. This plug-in is used to disassemble the machine code generated by the JIT compiler, offering a deeper understanding of how the Java code is being executed at the hardware level.

[12] https://github.com/async-profiler/async-profiler

[13] https://blogs.oracle.com/javamagazine/post/java-hotspot-hsdis-disassembler

Key Takeaways

In this section, we applied a top-down approach to understand and enhance the performance of our Java-centric OLTP system. We started at the highest system level, extending through the complex ecosystem of middleware, APIs, and storage engines. We then traversed the layers of the JVM and its runtime environment, crucial for efficient transaction processing. Our analysis incorporated the nuances of containerized environments. Understanding the JVM's behavior within these containers and how it interacts with orchestration platforms like Kubernetes is pivotal in aligning JVM resource allocation with the OLTP system's performance goals.

As we moved deeper, the role of the OS came into focus, especially in terms of process scheduling, memory management, and I/O operations. Here, we explored diagnostic tools to gain insights into how the OS influences the Java application, particularly in a high-transaction OLTP environment. Finally, we reached the foundational hardware subsystem, where we assessed the CPU, memory, storage, and network components. By employing system profilers and PMUs, we can gain a comprehensive view of the hardware's performance and its interaction with our software layers.

Performance engineering is a critical aspect of software engineering that addresses the efficiency and effectiveness of a system. It requires a deep understanding of each layer of the modern stack—from application down to hardware—and employs various methodologies and tools to identify and resolve performance bottlenecks. By continuously monitoring, judiciously profiling, expertly analyzing, and meticulously tuning the system, performance engineers can enhance the system's performance and scalability, thereby improving the overall user experience—that is, how effectively and efficiently the end user can interact with the system. Among other factors, the overall user experience includes how quickly the system responds to user requests, how well it handles high-demand periods, and how reliable it is in delivering the expected output. An improved user experience often leads to higher user satisfaction, increased system usage, and better fulfillment of the system's overall purpose.

But, of course, the journey doesn't end here. To truly validate and quantify the results of our performance engineering efforts, we need a robust mechanism that offers empirical evidence of our system's capabilities. This is where performance benchmarking comes into play.

The Importance of Performance Benchmarking

Benchmarking is an indispensable component of performance engineering, a field that measures and evaluates software performance. The process helps us ensure that the software consistently meets user expectations and performs optimally under various conditions. Performance benchmarking extends beyond the standard software development life cycle, helping us gauge and comprehend how the software operates, particularly in terms of its "ilities"—such as scalability, reliability, and similar nonfunctional requirements. Its primary function is to provide data-driven assurance about the performance capabilities of a system (i.e., the system under test [SUT]). Aligning with the methodologies we discussed earlier, the empirical approach provides us with assurance and validates theoretical underpinnings of system design with comprehensive testing and analysis. Performance benchmarking offers a reliable measure of how effectively and efficiently the system can handle varied conditions, ranging from standard operational loads to peak demand scenarios.

To accomplish this, we conduct a series of experiments designed to reveal the performance characteristics of the SUT under different conditions. This process is similar to how a unit of work (UoW) operates in performance engineering. For benchmarking purposes, a UoW could be a single user request, a batch of requests, or any task the system is expected to perform.

Key Performance Metrics

When embarking on a benchmarking journey, it's essential to extract specific metrics to gain a comprehensive understanding of the software's performance landscape:

- **Response time:** This metric captures the duration of each specific usage pattern. It's a direct indicator of the system's responsiveness to user requests.
- **Throughput:** This metric represents the system's capacity, indicating the number of operations processed within a set time frame. A higher throughput often signifies efficient code execution.
- **Java heap utilization:** This metric takes a deep dive into memory usage patterns. Monitoring the Java heap can reveal potential memory bottlenecks and areas for optimization.
- **Thread behavior:** Threads are the backbone of concurrent execution in Java. Analyzing counts, interactions, and potential blocking scenarios reveals insights into concurrency issues, including, starvation, over-and-under-provisioning, and potential deadlocks.
- **Hardware and OS pressures:** Beyond the JVM, it's essential to monitor system-level metrics. Keep an eye on CPU usage, memory allocation, network I/O, and other vital components to gauge the overall health of the system.

Although these foundational metrics provide a solid starting point, specific benchmarks often require additional JVM or application-level statistics tailored to the application's unique characteristics. By adhering to the performance engineering process and effectively leveraging both the top-down and bottom-up methodologies, we can better contextualize these metrics. This structured approach not only aids in targeted optimizations within our UoW, but also contributes to the broader understanding and improvement of the SUT's overall performance landscape. This dual perspective ensures a comprehensive optimization strategy, addressing specific, targeted tasks while simultaneously providing insights into the overall functioning of the system.

The Performance Benchmark Regime: From Planning to Analysis

Benchmarking goes beyond mere speed or efficiency measurements. It's a comprehensive process that delves into system behavior, responsiveness, and adaptability under different workloads and conditions. The benchmarking process involves several interconnected activities.

The Performance Benchmarking Process

The benchmarking journey begins with the identification of a need or a question and culminates in a comprehensive analysis. It involves the following steps (illustrated in Figure 5.12):

- **Requirements gathering:** We start by identifying the performance needs or asks guiding the benchmarking process. This phase often involves consulting the product's design to derive a performance test plan tailored to the application's unique requirements.
- **Test planning and development:** The comprehensive plan is designed to track requirements, measure specific implementations, and ensure their validation. The plan distinctly outlines the benchmarking strategy in regard to how to benchmark the unit of test (UoT), focusing on its limits. It also characterizes the workload/benchmark while defining acceptance criteria for the results. It's not uncommon for several phases of this process to be in progress simultaneously, reflecting the dynamic and evolving nature of performance engineering. Following the planning phase, we move on to the test development phase, keeping the implementation details in mind.
- **Performance Validation:** In the validation phase, every benchmark undergoes rigorous assessment to ensure its accuracy and relevance. Robust processes are put in place to enhance the repeatability and stability of results across multiple runs. It's crucial to understand both the UoT and the test harness's performance characteristics. Also, continuous validation becomes a necessary investment to ensure the benchmark remains true to its UoW and reflective of the real-world scenarios.

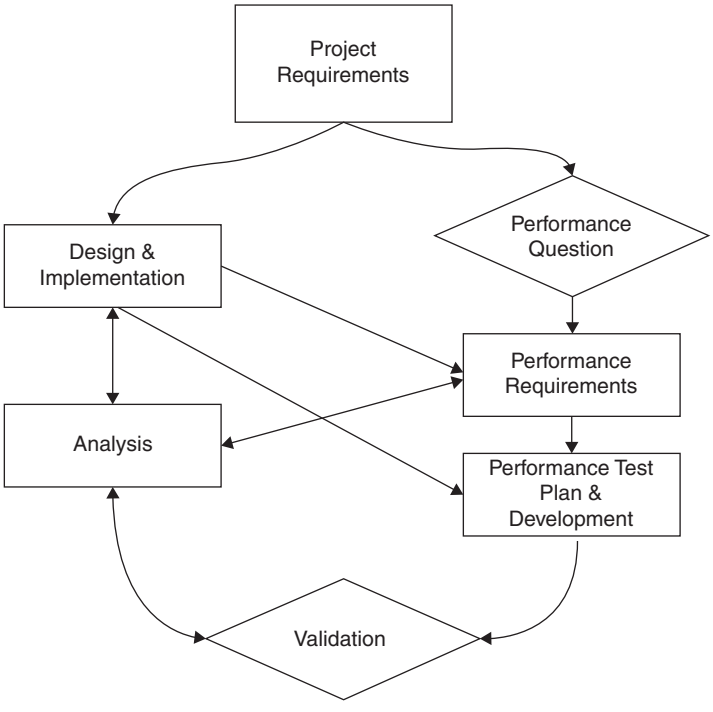

Figure 5.12 Benchmarking Process

- **Analysis:** In the analysis phase, we ensure that the test component (that is, the UoT) is stressed enough for performance engineers and developers to make informed decisions. This mirrors how performance engineers consider the interaction between the SUT and the UoW in performance evaluation. The accuracy of results is paramount, as the alternative is having irrelevant results drive our decision making. For instance, high variance in measurements indicates interference, necessitating an investigation into its source. Histograms of the measures, especially in high-variance scenarios, can be insightful. Performance testing inherently focuses on performance, presupposing that the UoT is functionally stable. Any performance test resulting in functional failures should be deemed a failed run, and its results discarded.

Iterative Approach to Benchmarking

Just as product design is continually revised and updated based on new insights and changing requirements, the performance plan is a living document. It evolves in response to the continuous insights gained through the performance analysis and design phases. This iterative process allows us to revisit, rethink, and redo any of the previous phases to take corrective action, ensuring that our benchmarks remain relevant and accurate. It enables us to capture system-level bottlenecks, measure resource consumption, understand execution details (for example, data structure locations on the heap), cache friendliness, and more. Iterative benchmarking also allows us to evaluate whether a new feature meets or exceeds the performance requirements without causing any regression.

When comparing two UoTs, it's essential to validate, characterize, and tune them independently, ensuring a fair comparison. Ensuring the accuracy and relevance of benchmarks is pivotal to making informed decisions in the realm of software performance.

Benchmarking JVM Memory Management: A Comprehensive Guide

In the realm of JVM performance, memory management stands out as a cornerstone. With the evolution of groundbreaking projects like Panama and Valhalla, and the advent of GCs such as G1, ZGC, and Shenandoah, the intricacies of JVM memory management have become a big concern. This section takes an in-depth look into benchmarking JVM memory management, a process that equips developers with the knowledge to optimize their applications effectively.

Setting Clear Objectives

- **Performance milestones:** Clearly define your goals. Whether it's minimizing latency, boosting throughput, or achieving optimal memory utilization, a well-defined objective will serve as the compass for your benchmarking endeavors.
- **Benchmarking environment:** Always benchmark within an environment that replicates your production setting. This ensures that the insights you gather are directly applicable to real-world scenarios.

Key Metrics to Benchmark

- **Allocation rates:** Monitor how swiftly your application allocates memory. Elevated rates can be indicative of inefficiencies in your application's memory usage patterns. With the OpenJDK GCs, high allocation rates and other stressors can combine forces and send the system into "graceful degradation" mode. In this mode, the GC may not meet its optimal pause time goals or throughput targets but will continue to function, ensuring that the application remains operational. Graceful degradation is a testament to the resilience and adaptability of these GCs, emphasizing the importance of monitoring and optimizing allocation rates for consistent application performance.
- **Garbage collection metrics:** It's essential to closely monitor GC efficiency metrics such as GC pause times, which reflect the durations for which application threads are halted to facilitate memory reclamation. Extended or frequent pauses can adversely affect application responsiveness. The frequency of GC events can indicate the JVM's memory pressure levels, with higher frequencies potentially pointing to memory challenges. Understanding the various GC phases, especially in advanced collectors, can offer granular insights into the GC process. Additionally, be vigilant of GC-induced throttling effects, which represent the performance impacts on the application due to GC activities, especially with the newer concurrent collectors.
- **Memory footprint:** Vigilantly track the application's total memory utilization within the JVM. This includes the heap memory, the segmented code cache, and the Metaspace. Each area has its role in memory management, and understanding their usage patterns can reveal optimization opportunities. Additionally, keep an eye on off-heap memory, which encompasses memory allocations outside the standard JVM heap and the JVM's own operational space, such as allocations by native libraries, thread stacks, and direct buffers. Monitoring this memory is vital, as unchecked off-heap usage can lead to unexpected out-of-memory or other suboptimal, resource-intensive access patterns, even if the heap seems to be comfortably underutilized.
- **Concurrency dynamics:** Delve into the interplay of concurrent threads and memory management, particularly with GCs like G1, ZGC, and Shenandoah. Concurrent data structures, which are vital for multithreaded memory management, introduce overheads in ensuring thread safety and data consistency. Maintenance barriers, which are crucial for memory ordering, can subtly affect both read and write access patterns. For instance, write barriers might slow data updates, whereas read barriers can introduce retrieval latencies. Grasping these barriers' intricacies is essential, as they can introduce complexities not immediately apparent in standard GC pause metrics.
- **Native memory interactions (for Project Panama):** Embracing the advancements brought by Project Panama represents an area of significant opportunity for JVM developers. By leveraging the Foreign Function and Memory (FFM) API, developers can tap into a new era of seamless and efficient Java-native interactions. Beyond just efficiency, Panama offers a safer interface, reducing the risks associated with traditional JNI methods. This transformative approach not only promises to mitigate commonly experienced issues such as buffer overflows, but also ensures type safety. As with any opportunity, it's crucial to benchmark and validate these interactions in real-world

scenarios, ensuring that you fully capitalize on the potential benefits and guarantees Panama brings to the table.

- **Generational activity and regionalized work:** Modern GCs like G1, Shenandoah, and ZGC employ a regionalized approach, partitioning the heap into multiple regions. This regionalization directly impacts the granularity of maintenance structures, such as remembered sets. Objects that exceed region size thresholds, termed "humongous," can span multiple regions and receive special handling to avert fragmentation. Additionally, the JVM organizes memory into distinct generations, each with its own collection dynamics. Maintenance barriers are employed across both generations and regions to ensure data integrity during concurrent operations. A thorough understanding of these intricacies is pivotal for advanced memory optimization, especially when analyzing region occupancies and generational activities.

Choosing the Right Tools

- **Java Micro-benchmarking Harness (JMH):** A potent tool for micro-benchmarks, JMH offers precise measurements for small code snippets.
- **Java Flight Recorder and JVisualVM:** These tools provide granular insights into JVM internals, including detailed GC metrics.
- **GC logs:** Analyzing GC logs can spotlight potential inefficiencies and bottlenecks. By adding visualizers, you can get a graphical representation of the GC process, making it easier to spot patterns, anomalies, or areas of concern.

Set Up Your Benchmarking Environment

- **Isolation:** Ensure your benchmarking environment is devoid of external interferences, preventing any distortions in results.
- **Consistency:** Uphold uniform configurations across all benchmarking sessions, ensuring results are comparable.
- **Real-world scenarios:** Emulate real-world workloads for the most accurate insights, incorporating realistic data and request frequencies.

Benchmark Execution Protocol

- **Warm-up ritual:** Allow the JVM to reach its optimal state, ensuring JIT compilation and class loading have stabilized before collecting metrics.
- **Iterative approach:** Execute benchmarks repeatedly, with results analyzed using geometric means and confidence intervals to ensure accuracy and neutralize outliers and anomalies.
- **Monitoring external factors:** Keep an eye on external influences such as OS-induced scheduling disruptions, I/O interruptions, network latency, and CPU contention. Be particularly attentive to the 'noisy neighbor' effect in shared environments, as these can significantly skew benchmark results.

Analyze Results

- **Identify bottlenecks:** Pinpoint areas where memory management might be hampering performance.
- **Compare with baselines:** If you've made performance-enhancing changes, contrast the new results with previous benchmarks to measure improvements.
- **Visual representation:** Utilize graphs for data trend visualization.

The Refinement Cycle

- **Refine code:** Modify your code or configurations based on benchmark insights.
- **Re-benchmark:** After adjustments, re-execute benchmarks to gauge the impact of those changes.
- **Continuous benchmarking:** Integrate benchmarking into your development routine for ongoing performance monitoring.

Benchmarking JVM memory management is an ongoing endeavor of measurement, analysis, and refinement. With the right strategies and tools in place, developers can ensure their JVM-based applications are primed for efficient memory management. Whether you're navigating the intricacies of projects like Panama and Valhalla or trying to understand modern GCs like G1, ZGC, and Shenandoah, a methodical benchmarking strategy is the cornerstone of peak JVM performance. Mastering JVM memory management benchmarking equips developers with the tools and insights needed to ensure efficient and effective application performance.

As we've delved into the intricacies of benchmarking and JVM memory management, you might have wondered about the tools and frameworks that can facilitate this process. How can you ensure that your benchmarks are accurate, consistent, and reflective of real-world scenarios? This leads us to the next pivotal topic.

Why Do We Need a Benchmarking Harness?

Throughout this chapter, we have emphasized the symbiotic relationship between hardware and software, including how their interplay profoundly influences their performance. When benchmarking our software to identify performance bottlenecks, we must acknowledge this dynamic synergy. Benchmarking should be viewed as an exploration of the entire system, factoring in the innate optimizations across different levels of the Java application stack.

Recall our previous discussions of the JVM, garbage collection, and concepts such as JIT compilation, tiered compilation thresholds, and code cache sizes. All of these elements play a part in defining the performance landscape of our software. For instance, as highlighted in the hardware–software interaction section, the processor cache hierarchies and processor memory model can significantly influence performance. Thus, a benchmark designed to gauge performance across different cloud architectures (for example) should be sensitive to these factors across the stack, ensuring that we are not merely measuring and operating in silos, but rather are taking full advantage of the holistic system and its built-in optimizations.

This is where a benchmarking harness comes into the picture. A benchmarking harness provides a standardized framework that not only helps us to circumvent common benchmarking pitfalls, but also streamlines the build, run, and timing processes for benchmarks, thereby ensuring that we obtain precise measurements and insightful performance evaluations. Recognizing this need, the Java community—most notably, Oracle engineers—has provided its own open-source benchmarking harness, encompassing a suite of micro-benchmarks, as referenced in JDK Enhancement Proposal (JEP) 230: *Microbenchmark Suite*.[14] Next, we will examine the nuances of this benchmarking harness and explore its importance in achieving reliable performance measurements.

The Role of the Java Micro-Benchmark Suite in Performance Optimization

Performance optimization plays a critical role in the release cycle, especially the introduction of features intended to boost performance. For instance, JEP 143: *Improve Contended Locking*[15] employed 22 benchmarks to gauge performance improvements related to object monitors. A core component of performance testing is regression analysis, which compares benchmark results between different releases to identify nontrivial performance regressions.

The Java Microbenchmark Harness (JMH) brings stability to the rigorous benchmarking process that accompanies each JDK release. JMH, which was first added to the JDK tree with JDK 12, provides scripts and benchmarks that compile and run for both current and previous releases, making regression studies easier.

Key Features of JMH for Performance Optimization

- **Benchmark modes:** JMH supports various benchmarking modes, such as Average Time, Sample Time, Single Shot Time, Throughput, and All. Each of these modes provides a different perspective on the performance characteristics of the benchmark.
- **Multithreaded benchmarks:** JMH supports both single-threaded and multithreaded benchmarks, allowing performance testing under various levels of concurrency.
- **Compiler control:** JMH offers mechanisms to ensure the JIT compiler doesn't interfere with the benchmarking process by over-optimizing or eliminating the benchmarking code.
- **Profilers:** JMH comes with a simple profiler interface plus several default profilers, including the GC profiler, Compiler profiler, and Classloader profiler. These profilers can provide more in-depth information about how the benchmark is running and where potential bottlenecks are.

JMH is run as a stand-alone project, with dependencies on user benchmark JAR files. For those readers who are unfamiliar with Maven templates, we'll begin with an overview of Maven and the information required to set up the benchmarking project.

[14] https://openjdk.org/jeps/230

[15] https://openjdk.org/jeps/143

Getting Started with Maven

Maven is a tool for facilitating the configuration, building, and management of Java-based projects. It standardizes the build system across all projects through a project object model (POM), bringing consistency and structure to the development process. For more information about Maven, refer to the Apache Maven Project website.[16]

After installing Maven, the next step is setting up the JMH project.

Setting Up JMH and Understanding Maven Archetype

The Maven Archetype is a project templating toolkit used to create a new project based on the JMH project Archetype. Its setup involves assigning a "GroupId" and an "ArtifactId" to your local project. The JMH setup guide provides recommendations for the Archetype and your local project. In our example, we will use the following code to generate the project in the `mybenchmarks` directory:

```
$ mvn archetype:generate \
   -DarchetypeGroupId=org.openjdk.jmh \
   -DarchetypeArtifactId=jmh-java-benchmark-archetype \
   -DarchetypeVersion=1.36 \
   -DgroupId=org.jmh.suite \
   -DartifactId=mybenchmarks \
   -DinteractiveMode=false \
   -Dversion=1.0
```

After running the commands, you should see a "BUILD SUCCESS" message.

Writing, Building, and Running Your First Micro-benchmark in JMH

The first step in creating a micro-benchmark is to write the benchmark class. This class should be located in the `mybenchmarks` directory tree. To indicate that a method is a benchmark method, annotate it with `@Benchmark`.

Let's write a toy benchmark to measure the performance of two key operations in an educational context: registering students and adding courses. The `OnlineLearningPlatform` class serves as the core of this operation, providing methods to register students and add courses. Importantly, the `Student` and `Course` entities are defined as separate classes, each encapsulating their respective data. The benchmark methods `registerStudentsBenchmark` and `addCoursesBenchmark` in the `OnlineLearningPlatformBenchmark` class aim to measure the performance of these operations. The simple setup offers insights into the scalability and performance of the `OnlineLearningPlatform`, making it an excellent starting point for beginners in JMH.

```
package org.jmh.suite;

import org.openjdk.jmh.annotations.Benchmark;
```

[16] https://maven.apache.org/

```
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Scope;

// JMH Benchmark class for OnlineLearningPlatform
@State(Scope.Benchmark)
public class OnlineLearningPlatformBenchmark {

    // Instance of OnlineLearningPlatform to be used in the benchmarks
    private final OnlineLearningPlatform platform = new OnlineLearningPlatform();

    // Benchmark for registering students
    @Benchmark
    public void registerStudentsBenchmark() {
        Student student1 = new Student("Annika Beckwith");
        Student student2 = new Student("Bodin Beckwith");
        platform.registerStudent(student1);
        platform.registerStudent(student2);
    }

    // Benchmark for adding courses
    @Benchmark
    public void addCoursesBenchmark() {
        Course course1 = new Course("Marine Biology");
        Course course2 = new Course("Astrophysics");
        platform.addCourse(course1);
        platform.addCourse(course2);
    }

}
```

The directory structure of the project would look like this:

```
mybenchmarks
└──src
    └──main
        └──java
            └──org
                └──jmh
                    └──suite
                        └──OnlineLearningPlatformBenchmark.java
                        └──OnlineLearningPlatform.java
                        └──Student.java
                        └──Course.java
target
└──benchmarks.jar
```

Once the benchmark class is written, it's time to build the benchmark. The process remains the same even when more benchmarks are added to the `mybenchmarks` directory. To build and install the project, use the Maven command `mvn clean install`. After successful execution, you'll see the "BUILD SUCCESS" message. This indicates that your project has been successfully built and the benchmark is ready to be run.

Finally, to run the benchmark, enter the following at the command prompt:

```
$ java –jar target/benchmarks.jar org.jmh.suite.OnlineLearningPlatformBenchmark
```

The execution will display the benchmark information and results, including details about the JMH and JDK versions, warm-up iterations and time, measurement iterations and time, and others.

The JMH project then presents the aggregate information after completion:

```
# Run complete. Total time: 00:16:45

Benchmark                                                  Mode   Cnt Score          Error        Units
OnlineLearningPlatformBenchmark.registerStudentsBenchmark thrpt  25  50913040.845 ± 1078858.366  ops/s
OnlineLearningPlatformBenchmark.addCoursesBenchmark       thrpt  25  50742536.478  ± 1039294.551 ops/s
```

That's it! We've completed a run and found our methods executing a certain number of operations per second (default = Throughput mode). Now you're ready to refine and add more complex calculations to your JMH project.

Benchmark Phases: Warm-Up and Measurement

To ensure developers get reliable benchmark results, JMH provides `@Warmup` and `@Measurement` annotations that help specify the warm-up and measurement phases of the benchmark. This is significant because the JVM's JIT compiler often optimizes code during the first few runs.

```
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5)
@Measurement(iterations = 5)
public class MyBenchmark {

    @Benchmark
    public void testMethod() {
        // your benchmark code here...
    }
}
```

In this example, JMH first performs five warm-up iterations, running the benchmark method but not recording the results. This warm-up phase is analogous to the ramp-up phase in the application's life cycle, where the JVM is gradually improving the application's performance until it reaches peak performance. The warm-up phase in JMH is designed to reach this steady-state before the actual measurements are taken.

After the warm-up, JMH does five measurement iterations, where it does record the results. This corresponds to the steady-state phase of the application's life cycle, where the application executes its main workload at peak performance.

In combination, these features make JMH a powerful tool for performance optimization, enabling developers to better understand the behavior and performance characteristics of their Java code. For detailed learning, it's recommended to refer to the official JMH documentation and sample codes provided by the OpenJDK project.[17]

Loop Optimizations and @OperationsPerInvocation

When dealing with a loop in a benchmark method, you must be aware of potential loop optimizations. The JVM's JIT compiler can recognize when the result of a loop doesn't affect the program's outcome and may remove the loop entirely during optimization. To ensure that the loop isn't removed, you can use the `@OperationsPerInvocation` annotation. This annotation tells JMH how many operations your benchmark method is performing, preventing the JIT compiler from skipping the loop entirely.

The `Blackhole` class is used to "consume" values that you don't need but want to prevent from being optimized away. This outcome is achieved by passing the value to the `blackhole.consume()` method, which ensures the JIT compiler won't optimize away the code that produced the value. Here's an example:

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@OperationsPerInvocation(10000)
public void measureLoop() {
    for (int i = 0; i < 10000; i++) {
        blackhole.consume(i);
    }
}
```

In this example, the `@OperationsPerInvocation(10000)` annotation informs JMH that the `measureLoop()` method performs 10,000 operations. The `blackhole.consume(i)` call ensures that the loop isn't removed during JIT optimizations, leading to accurate benchmarking of the loop's performance.

[17] https://github.com/openjdk/jmh

Benchmarking Modes in JMH

JMH provides several benchmarking modes, and the choice of mode depends on which aspect of performance you're interested in.

- **Throughput:** Measures the benchmark method's throughput, focusing on how many times the method can be called within a given time unit. This mode is useful when you want to measure the raw speed, such as operations in a data stream.
- **Average Time:** Measures the average time taken per benchmark method invocation, offering insights into how long it typically takes to execute a single method call. This mode is useful for understanding the performance of discrete operations, such as processing a request to a web server.
- **Sample Time:** Measures the time it takes for each benchmark method invocation and calculates the distribution of invocation times. This mode is useful when you want to understand the distribution of times, not just the average, such as latency of a network call.
- **Single Shot Time:** Measures the time it takes for a single invocation of the benchmark method. This mode is useful for benchmarks that take a significant amount of time to execute, such as a complex algorithm or a large database query.
- **All:** Runs the benchmark in all modes and reports the results for each mode. This mode is useful when you want a comprehensive view of your method's performance.

To specify the benchmark mode, use the `@BenchmarkMode` annotation on your benchmark method. For instance, to benchmark a method in throughput mode, you would use the following code:

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void myMethod() {
    // ...
}
```

Understanding the Profilers in JMH

In addition to supporting the creation, building, and execution of benchmarks, JMH provides inbuilt profilers. Profilers can help you gain a better understanding of how your code behaves under different circumstances and configurations. In the case of JMH, these profilers provide insights into various aspects of the JVM, such as garbage collection, method compilation, class loading, and code generation.

You can specify a profiler using the `-prof` command-line option when running the benchmarks. For instance, `java -jar target/benchmarks.jar -prof gc` will run your benchmarks with the GC profiler enabled. This profiler reports the amount of time spent in GC

and the total amount of data processed by the GC during the benchmarking run. Other profilers can provide similar insights for different aspects of the JVM.

Key Annotations in JMH

JMH provides a variety of annotations to guide the execution of your benchmarks:

- **`@Benchmark`:** This annotation denotes a method as a benchmark target. The method with this annotation will be the one where the performance metrics are collected.
- **`@BenchmarkMode`:** This annotation specifies the benchmarking mode (for example, Throughput, Average Time). It determines how JMH runs the benchmark and calculates the results.
- **`@OutputTimeUnit`:** This annotation specifies the unit of time for the benchmark's output. It allows you to control the time granularity in the benchmark results.
- **`@Fork`:** This annotation specifies the number of JVM forks for each benchmark. It allows you to isolate each benchmark in its own process and avoid interactions between benchmarks.
- **`@Warmup`:** This annotation specifies the number of warm-up iterations, and optionally the amount of time that each warm-up iteration should last. Warm-up iterations are used to get the JVM up to full speed before measurements are taken.
- **`@Measurement`:** This annotation specifies the number of measurement iterations, and optionally the amount of time that each measurement iteration should last. Measurement iterations are the actual benchmark runs whose results are aggregated and reported.
- **`@State`:** This annotation is used to denote a class containing benchmark state. The state instance's life-cycle[18] is managed by JMH. It allows you to share common setup code, variables and define thread-local or shared scope to control state sharing between benchmark methods.
- **`@Setup`:** This annotation marks a method that should be executed to set up a benchmark or state object. It's a life-cycle method that is called before the benchmark method is executed.
- **`@TearDown`:** This annotation marks a method that should be executed to tear down a benchmark or state object. It's a life-cycle method that is called after the benchmark method is executed.
- **`@OperationsPerInvocation`:** This annotation indicates the number of operations a benchmark method performs in each invocation. It's used to inform JMH about the number of operations in a benchmark method, which is useful to avoid distortions from loop optimizations.

[18] 'Lifecycle' refers to the sequence of stages (setup, execution, teardown) a benchmark goes through.