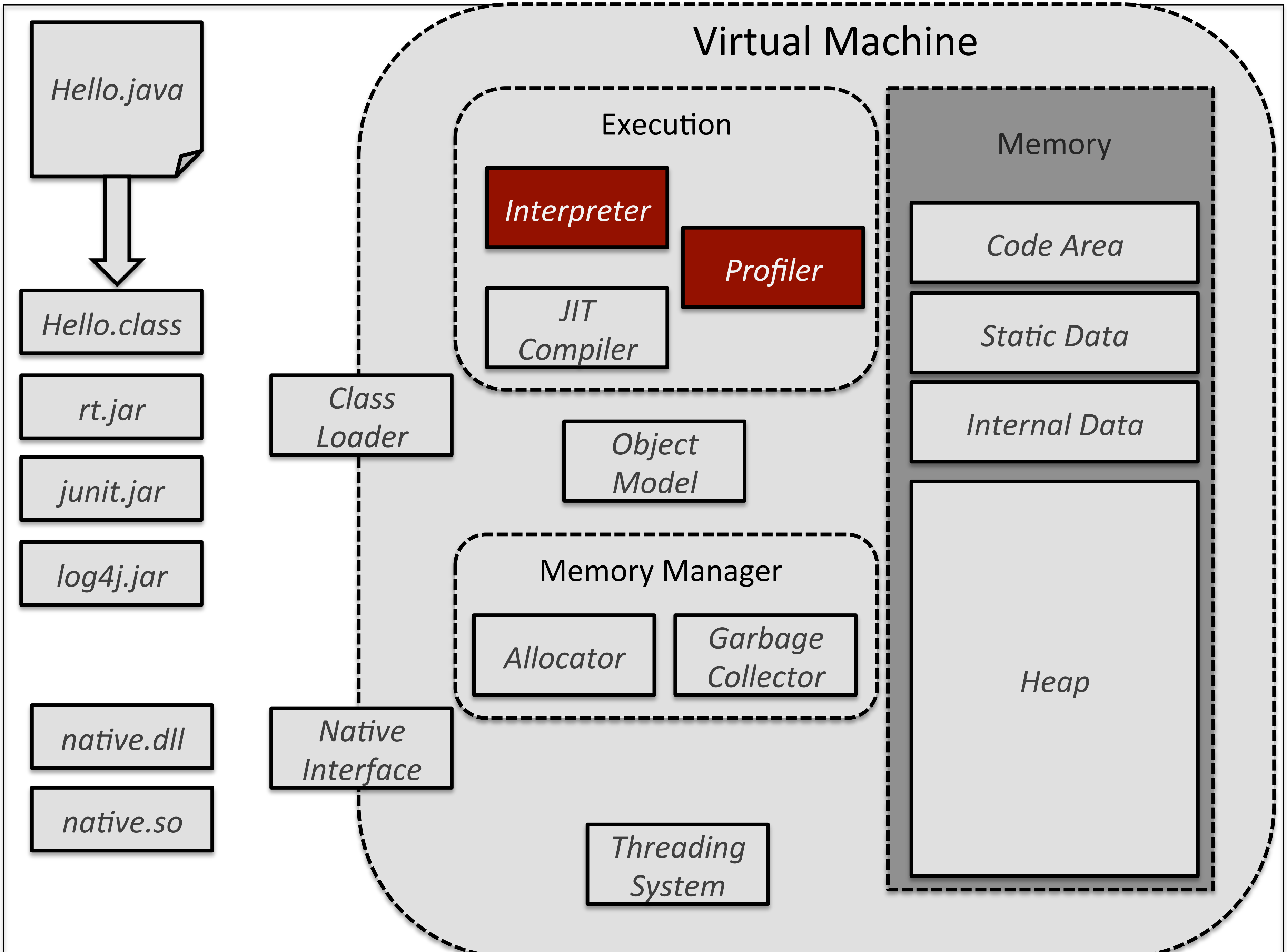
Dynamic Profiling



Virtual Machine
Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Class Loader
Native Interface
Execution
Interpreter
Profiler
JIT Compiler
Object Model
Memory Manager
Allocator
Garbage Collector
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Ahead of Time Compilation

- Code is compiled before being shipped
- Compilation overhead paid ahead of time
  - Applications start up fast
- Longer development cycle
  - Compile overhead can be minimized
- Code produced is platform-specific
  - Depends on architecture, OS, libraries
  - Remember Apple's fat binaries

Just In Time

- Named after manufacturing technique
  - Started in 1950s Japanese auto industry
  - Companies don't want to maintain warehouses
  - Arrange for parts to be delivered when needed
  - Signals and feedback to get flow right

- Compiler technique is similar
  - Popularized in the 60s
  - Minimize waste and overhead
  - Defer work as long as possible

JIT Compilers

- Perform compilation at runtime
  - Code compiled only when it is needed
  - Some code is never executed, so never compiled
- Compiled code cached
  - Saves on repeated compilation work
- Ultimately does a similar amount of work
  - Defers compile overhead to run time

Mixed Mode Execution

- Interpreted and compiled code can coexist
  - Simulated stack machine for interpretation
  - Native code for compiled
- Not a trivial engineering task
  - Data and control must flow between modes
  - Stack frames may alternate modes
- Not all code has to be compiled on demand
  - Can defer compilation to a better time
  - Can compile in the background
- Modes don't have to be interpreter vs JIT
  - Can use a fast initial compiler

Guided Optimization

- We don't have to compile all of the code
  - Important code can be optimized
  - Everything else interpreted
- Need to figure out what is important
  - Compilation vs execution time
- Compile hot code and use that next time
  - Next method invocation
  - Next pass around a loop

Profiling Data

- Classes loaded
- Methods called
- Loops executed
- Branches followed, code paths executed
- Invarients – types, constants, nulls
- Profile data improves over time

Hot Methods

- Not all methods are called equally
  - Some methods are called only once
  - Others called frequently
- Profiler keeps a count of method invocations
  - Can be fast
  - Can use register overflow to check for a limit
- Programs tend to have small working sets
  - Although this may change over time
  - Phase behavior

Loops

- Recall how a loop was structured
  - Unconditional jump (GOTO) over the body
  - Load values for comparison
  - Jump backward depending on the result
- Back branches are more interesting
  - Indicate how many times the loop is required
- Good way to detect a loop

Call Sites

- Place from which a method is called
- Simplest call sites are monomorphic
  - Exactly one method is called from the site
- More methods called from a polymorphic site
  - Multiple methods are called from the same site
- Can also differentiate bimorphic or megamorphic

```java
public int getRandomHashCode(final List<Object> lst) {
    final int size = lst.size();
    final double random = Math.random();
    final int idx = (int) (size * random);
    final Object obj = lst.get(idx);
    return obj.hashCode();
}
```

```
public int getRandomHashCode(final List<Object> lst) {
    final int size = lst.size();
    final double random = Math.random();
    final int idx = (int) (size * random);
    final Object obj = lst.get(idx);
    return obj.hashCode();
}
```

Polymorphic

```java
public int getRandomHashCode(final List<Object> lst) {
    final int size = lst.size();
    final double random = Math.random();
    final int idx = (int) (size * random);
    final Object obj = lst.get(idx);
    return obj.hashCode();
}
```

Polymorphic

Monomorphic

```java
public int getRandomHashCode(final List<Object> lst) {

    final int size = lst.size();                      Polymorphic

    final double random = Math.random();              Monomorphic

    final int idx = (int) (size * random);

    final Object obj = lst.get(idx);                  Polymorphic

    return obj.hashCode();
}
```

```
public int getRandomHashCode(final List<Object> lst) {

    final int size = lst.size();                             Polymorphic

    final double random = Math.random();                     Monomorphic

    final int idx = (int) (size * random);

    final Object obj = lst.get(idx);                         Polymorphic

    return obj.hashCode();                                   Megamorphic
}
```

Call Sites

- Monomorphic are easiest to optimize
  - We know exactly what the control flow will be
- Others can be optimized
  - Takes more analysis
  - Guesses may be wrong

Class Hierarchy Analysis

- Some optimizations need a big picture view
  - Inheritance structure for a given class
  - All possible implementations of a method
- Straightforward in an AOT compiler
  - Scan all code files
- More difficult in a runtime system
  - Dynamic class loading

Class Hierarchy Analysis

- Snapshot all currently loaded classes
- Calculate inheritance graph
- Calculate method overriding
- Must be updated on classloading
  - Update can be incremental

Profile-Guided Optimization

- Can be an educated guess
  - Guesses can be wrong
- Sometimes we know for sure
  - final keyword tells us that things won’t change
- There are cases when we are optimistic
  - Need a fall-back in case we are wrong

Persistent Profiling

- Profiling data can be stored across runs
  - Assume that subsequent executions are similar
- Use previous execution as a hint
- Not widely used outside of research