Initializing the threading system is an important part of the `VM.boot` method. It creates the necessary garbage-collection threads, a thread for running object finalizer methods, and threads responsible for the adaptive optimization system. A debugger thread is also created, but is scheduled for execution only if a signal is sent to Jikes RVM from the operating system. The final thread to be created and to start execution is the main thread, which is responsible for running the Java application.

Runtime Components

The previous section described getting Jikes RVM to a point where it is ready for execution. In this section, we look at the main runtime components of Jikes RVM, beginning with those directly responsible for executing Java bytecode, and then look at some of the other virtual machine subsystems that support this execution.

Basic Execution Model

Jikes RVM does not include an interpreter; all bytecodes must first be translated by one of Jikes RVM's compilers into native machine code. The unit of compilation is the method, and methods are compiled lazily when they are first invoked by the program. This initial compilation is done by Jikes RVM's *baseline compiler*, a simple nonoptimzing compiler that generates low-quality code very quickly. As execution continues, Jikes RVM's *adaptive system* monitors program execution to detect program hot spots and selectively recompiles them with Jikes RVM's optimizing compiler. This is a significantly more sophisticated compiler that generates higher-quality code, but at a significantly larger cost in compile time and compiler memory footprint than the baseline compiler.

This selective optimization model is not unique to Jikes RVM. All modern production JVMs rely on some variant of selective optimization to target optimizing compilation resources to the subset of the program's methods where they will yield the most benefit. As discussed earlier, selective optimization is the key to enabling the deployment of sophisticated optimizing compilers as dynamic compilers.

Adaptive Optimization System

Architecturally, the Adaptive Optimization System is implemented as a collection of loosely synchronized entities. Because it is implemented in Java, we are able to utilize built-in language features such as threads and monitors to structure the code.

As the program executes, timer-based samples are accumulated by the running Java threads into sampling buffers. Two types of profile data are collected: samples of the currently executing method (to guide identification of candidate methods for optimizing compilation) and call-stack samples (to identify important call graph edges for profile-directed inlining). When a sampling buffer is full, the low-level profiling agent signals a higher-level *Organizer*