unboxed types are held are of the same interest to the garbage collector as a memory location that holds a primitive field, such as an `int`.

Having strongly typed access to memory isn't quite enough to allow Jikes RVM to run without issue. Some special methods are required that indicate to the compilers that either something intrinsic is occurring or they should shortcut some parts of Java's strong type semantics. These special methods exist in `org.jikesrvm.runtime.Magic`. An example of an intrinsic operation is the square root method, for which Java provides no bytecode but many computer architectures have an instruction, or the routines that directly access fields for reflection. An example of working around Java's strong semantics is to avoid casts during thread scheduling, as a runtime exception wouldn't be permissible at certain key points.

All magic operations are compiled differently than regular methods, and they can't be compiled as standalone methods. Methods exist in their place, but if these were ever executed, they'd fail with an exception. Instead, the compiler determines from the method which magic operation is being performed, and then it looks up what operations the compiler must provide for that. During boot image creation, replacements for some of the unboxed magic types and methods are provided. For example, addresses that are in the boot image are unknown until objects are laid out. The boot image unboxed types keep track of identifiers and which objects they map to. During boot image creation, these identifiers are linked to the object they reference.

Although magic operations and unboxed types allow Java code in Jikes RVM the access to memory that a pointer would have in C or C++, the pointers are strongly typed and cannot be used in place of references. Strong typing allows programmer errors to be detected at compile time, good IDE integration, and support from bug finding tools, all of which have aided the development of Jikes RVM.

## Thread Model

Java has integral threading support. In 1998, operating system support for threading was not well-suited for many Java applications. In particular, typical operating-system threading implementations did not scale to support hundreds of threads in a single process, and locking operations (required to implement Java synchronized methods) were prohibitively expensive. To avoid this overhead, some JVMs implemented threading themselves, using a mode of operation called *green threading*. In green threading, threads are implemented by having the JVM itself manage the process of multiplexing multiple Java threads onto a smaller number of operating system threads (often one, or one per CPU in a multiprocessor system). The primary advantage of green threading is that it allows the implementations of locking and thread scheduling that are specialized to Java semantics and that can be made highly scalable. The primary disadvantage is that the operating system is not aware of what the JVM is doing, which can lead to a number of performance pathologies, especially if the Java application is interacting heavily with native code (which may itself be making assumptions about the