principles and trusted benchmarks only when large shifts in performance occurred. In a just-in-time compiled environment, small changes in these micro benchmarks are, at best, not repeatable on a separate system. At worst, they are so highly dependent on the benchmarked scenario that they are not even reliably repeatable on the same system.

> NOTE
> **One important feature of the microcode set is that the integer values to which the constants are set are sequential. The core of the interpreter is a switch statement on this set of constants, and we need to ensure that this switch runs as fast as possible.**

With these factors in mind, we concluded after several experiments that a set with 750 or so codes for complete integer and floating-point emulation represented a good trade-off. This gives us an approximate factor of 10 conversion from x86 operation to microcodes. Although this set may seem large, it decodes quickly and the operations are reasonably atomic. This makes them good candidates to feed into the later optimizing stages.

TIP #3: TABLE SWITCH GOOD, LOOKUP SWITCH BAD

Switch statements whose labels are a reasonably compact set are faster than those whose values are more disparate. This is because Java has two bytecodes for switches: `tableswitch` and `lookupswitch`. Table switches are performed using an indirect call, with the switch value providing the offset into a function table. Lookup switches are much slower because they perform a map lookup to find a matching value:function pair.

Hijacking the JVM

The refrain of "Java is slow" haunts Java developers to this day. The bulk of this comment derives from the experiences of non-Java developers with early JVMs back in the mid-to-late 1990s. Since that time those of us who work with Java know that it has moved in leaps and bounds. The driving force behind this improvement is also the key to speeding up JPC: the environment of a conventional Java process can be partitioned very simply between program regions and data regions.

In Figure 9-6 we can see that the data region is then further split between static data, which can be known at compile time, and dynamic data, which cannot. The bulk of the static data in a Java environment are the class bytes loaded from the classpath. Although the class bytes are loaded as data, it is clear that they actually represent code, and they will get interpreted by the JVM at runtime. So it is obvious that we would like to maneuver these class bytes onto the other side of the diagram.