We then repeat the just-in-time compilation trick performed by the JVM, but do so at the JPC level. So for JPC we have a second-tier program-data information divide within the confines of the Java Runtime Environment. Our compilation now has two stages:

1. IA-32 machine code is compiled into bytecode on demand within JPC. These x86 blocks thus become valid Java class files that can be loaded as such by the JVM.
2. Classes are compiled by the JVM into native code. As the JVM does not distinguish between the original "static" classes that comprise the handwritten code and the dynamic classes built automatically, both types are optimized to get the best native performance possible.

After these two stages of compilation, our original IA-32 machine code will have been translated into the host machine's architecture (see Figure 9-8). With a nice dollop of good fortune, the new instruction count will not be significantly larger than the original, which means performance will not be significantly slower than native.

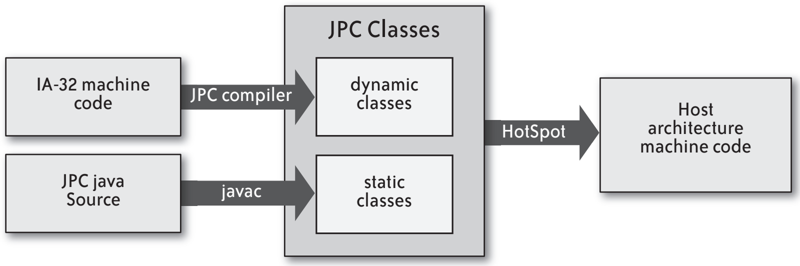

FIGURE 9-8. The three compilers in the JPC architecture

So now we know how to get more speed out of our emulator, but we have slightly glossed over the details. These details (our new problems) now fall into two distinct areas: how do we perform this compilation, and how should we load the resultant classes?

Compiling: How to Reinvent the Wheel

The compiler we describe here is not the most optimal example, but we are in a slightly unusual situation. Both *javac* and the JPC compiler are only first-stage compilers, meaning they simply feed their output into a second stage, be this a bytecode interpreter or a just-in-time compiler. We know that *javac* does very little to optimize its output; bytecodes output by *javac* are simply a translation of the input Java code. See Example 9-2.