2 registers, and 128 bytes of RAM. A simple program equivalent to the following C code was written in Toy assembly language for speed tests:

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 50; j++)
        memory[51 + j] += 4;
```

The memory is initially all zero, and the output of the program is the memory state at the end (the first 50 bytes of memory are reserved for the program code).

A Java emulation of the Toy architecture was built, and 100,000 sequential runs of this program on the Sun HotSpot VM took 8000 ms. On the same hardware, a GCC-compiled C program took 86 ms (compiled with all static optimizations). Not surprisingly, naive emulation suffered a performance penalty of two orders of magnitude. But this simple emulation was indeed very simple, reading the next assembly instruction each time from memory, selecting the operation to carry out via a switch statement, and looping around until complete.

A better version would be to eliminate this lookup-dispatch process. This represents the commonplace trick of inlining; a C compiler inlines often-used code at compile time and the HotSpot VM does this at runtime in response to real execution telemetry. This done, the emulator then took 800 ms, a 10 times speed improvement.

Now, when using lookup-dispatch, the instruction pointer must be updated after each instruction so that the processor knows where to fetch the next one. However, with the code inlined, the instruction pointer needs to be updated only when the whole inline block has been executed. Removing these interleaved increments also helps HotSpot optimize the code; it can focus on the key operations without worrying about the need to keep this instruction pointer register always consistent with progress. With the instruction pointer updates shifted to the end of the inlined program sections, the execution speed reduced to 250 ms, an additional improvement of 3.2 times.

Nevertheless, the assembly code, translated by a simplistic automatic algorithm, resulted in many unnecessary movements of data between memory and registers. In several places a result was stored into the byte array of memory and then immediately read out again. A simple flow control analysis meant that these inefficiencies can be detected automatically and removed, resulting in a execution speed of 80 ms. This is as fast as the optimized native program!

Thus, with suitable runtime compilation, not much of it particularly complicated, a Java emulator can run this Toy native code at 100% native hardware speed. Clearly, expecting 100% when the vastly more complex hardware of the x86 PC is examined would be unrealistic, but these tests did suggest that practically useful speed might be attainable.

So, despite what might have been initial skepticism for the whole concept of a pure Java x86 emulator, there was actually sufficient evidence that such technology could be built. The next sections detail how the architectural aspects of the x86 PC hardware design were exploited and mapped onto the JVM to make an efficient emulation. A number of critically important