
Some Java compilers use an optimization called **object inlining**. In the usual form of this optimization, if the compiler is certain that an object A is pointed to only once, from another object B, then A can be allocated as part of B’s heap record, rather than in its own, separate heap record. This saves the cost of dereferencing the pointer to A, and the heap allocation and garbage collection overheads associated with the second record.

Based on the ideas in the assigned paper by Chilimbi, Hill, and Larus, argue why object inlining is not necessarily a good idea. Sketch some Java code that might be made slower if object inlining were performed.

2. Adve and Boehm paper.

a. Consider Figure 1 of the paper, showing the core of Dekker’s algorithm. The text describes a plausible compiler transformation on this code that would yield a non-sequentially-consistent result. What is that transformation, why might a compiler use it, and exactly what would be the bad result?

b. Explain, with an example, how speculative execution of branch instructions can lead to “self-fulfilling” effects on memory.

3. Branch prediction.

Using whatever hardware platform is convenient for you, design and execute an experiment to measure the (approximate) penalty, in machine cycles, of a branch misprediction on that hardware. Submit your code and the results of your tests. A priori, we would expect the answer to be roughly equal to the pipeline depth of your processor. If you are able to find any published numbers for the pipeline depth or misprediction cost on your hardware, you should compare to them too.

Here’s one approach: Write a simple program that repeatedly (many millions or billions of times) executes a conditional branch instruction. The program should be parameterized by a boolean control flag allowing it to be run in either of two modes. In *predictable* mode, the branch always goes the same way (hence highly predictable); in *unpredictable* mode, the branch should be very unpredictable, so that the hardware prediction mechanism (no matter how fancy) will only predict correctly about 50% of the time. Time execution under each mode setting; if you design the program carefully, the difference in times should be due entirely to the cost of mispredictions. More precisely, if \( n \) is the repetition count for your tests, and \( c \) is your clockspeed (in Hz), then

\[
\text{misprediction cost in cycles} = \frac{\text{runtime}_{\text{unpredictable}} - \text{runtime}_{\text{predictable}}}{n} \times 2 \times c
\]

(The factor of 2 is because only about half your “unpredictable” tests will actually be mispredicted.)

The main challenge of this exercise is controlling for other factors that might affect runtime. Be sure to use a very large \( n \) so that the fixed code (outside your repetition loop) doesn’t make a significant contribution to runtimes. And make sure that the machine code you execute really does contain the branch instruction you expect!

Warning: I’ve done this experiment, and I don’t entirely believe the answers I’m getting. Published figures in respectable textbooks seem even less believable. So don’t obsess too much...

**How to submit your homework.**

Submit the homework by mail to apt@cs.pdx.edu with subject line “CS577 HW3.” Put the answers to all questions into a single *plain text* file and attach it to your mail.