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.

Tip #4: Small methods are good methods

Small chunks of code are nice, as just-in-time environments generally see code on a method granularity. A large method that contains a "hot" area, may be compiled in its entirety. The resultant larger native code may cause more code-cache misses, which is bad for performance.

Tip #5: Exceptions are exceptional

Exceptions should be used for exceptional conditions, not just for errors. Using exceptions for flow control in unusual circumstances provides a hint to the VM to optimize for the nonexception path, giving you optimal performance.

Tip #6: Use decorator patterns with care

The decorator pattern is nice from a design point of view, but the extra indirection can be costly. Remember that it is permitted to remove decorators as well as add them. This removal may be considered an "exceptional occurrence" and can be implemented with a specialized exception throw.

Tip #7: `instanceof` *is faster on classes*

Performing `instanceof` on a class is far quicker than performing it on an interface. Java's single inheritance model means that on a class, `instanceof` is simply one subtraction and one array lookup; on an interface, it is an array search.

Tip #8: Use `synchronized` *minimally*

Keep synchronized blocks to a minimum; they can cause unnecessary overhead. Consider replacing them with either atomic or volatile references if possible.

Tip #9: Beware external libraries

Avoid using external libraries that are overkill for your purposes. If the task is simple and critical, then seriously consider coding it internally; a tailor-made solution is likely to be better suited to the task, resulting in better performance and fewer external dependencies.

Four in Four: It Just Won't Go

Many of the problems that we have come across while developing and designing JPC have been those associated with overhead. When trying to emulate a full computing environment inside itself, the only things that prevent you from extracting 100% of the native performance are the overheads of the emulation. Some of these overheads are time related such as in the processor emulation, whereas others are space related.