MinimizingObjectOverhead

Abstract

Lucy must be designed so that native OO overhead is minimized.

- Algorithms which create and destroy as few native objects as possible are preferred.
- Native method calls should be avoided for high-volume, performance critical applications. Possible alternatives:
    - Change the algo to use a for/while loop (either native or C).
    - Use a C struct with a function pointer.
    - Use a C function.

Rationale

Several ports of Lucene have been written for interpreted languages such as Perl, Python, and Ruby. The "pure" ports, written entirely in native code, have all suffered from relatively disappointing execution speed.

The Perl project Plucene is one example. Plucene was written by a group of seasoned programmers, including prolific CPAN contributor Tony Bowden and Simon Cozens, author of several books on Perl including one on the language's C internals. They expected that there would be some amount of performance degradation at launch, but planned to identify bottlenecks through profiling and optimize later.

Unfortunately, the sluggishness proved both more significant and more tenacious than anticipated. Many benchmarks were run and many patches were applied, but gains were incremental rather than order-of-magnitude as required. Gradually, opportunities for optimization were exhausted, and it became clear that the problem was fundamental and architectural. As Tony wrote in a message to the Plucene mailing list...

> We've done a LOT of benchmarking and profiling and the like, and we've been left with the rather unsatisfying feeling that one of the main speed issues is just the huge disparity in the speed of method calls in Java vs Perl. In Java method calls are pretty much free, so even in something heavily optimised for performance, such as Lucene, can have gazillions of them, even in tight loops. In Perl, OTOH, they're really expensive, comparatively, and so 'translating' a lot of the Lucene code straight into Perl plays straight into this weakness.

(Subsequent study revealed that overhead from object creation/destruction was an even more significant factor than method call overhead).

When Kino{{`Search was rewritten as a "loose port" of Lucene in Perl and C (it had originally been a completely independent project), the insights yielded by the Plucene team's dogged experimentation informed many of the design decisions. It would have been much more difficult for Kino}}`Search to achieve performance comparable to Lucene's without their efforts, and so Kino`Search, and by extension, Lucy, owes a debt of gratitude to Plucene's authors.