Object-oriented Performance

One of the traditional reasons for resisting the move from a legacy procedural programming style to an object-oriented programming style is runtime performance. Initially, refactoring monolithic procedural code into a group of small, individual classes and methods can introduce some performance regressions into a product. Object-oriented design principles allow programs to reuse more code, however. Reuse of code means fewer lines of code, which translates to shorter development time and fewer defects, allowing a greater focus on performance and new features. Furthermore, optimizations on a widely used method can benefit not only the original caller, but all callers that use it. This article reveals just how BBj® 9.0 used some of these very same principles in the BASIS codebase to provide significant performance enhancements to object-oriented BBj code.

By Adam Hawthorne
Software Engineer

Analysis

Amdahl's law is a theoretical statement about program optimization. One form of it argues that optimizing the portion of a computation that takes the longest amount of time produces the greatest performance gain.

Consider the implementation of some program operation. Each line of code in that implementation represents a potential optimization candidate. When a program spends all its time in one small section of code, the optimal candidate for optimization becomes obvious. But when there are so many lines of code or logical sections of code that each one only contributes a very small percentage of the total, the return on the effort required for any individual optimization project will seldom be compelling.

This is where object-oriented design can help. First, organizing operations into objects helps to limit the scope of an optimization. Optimizations can occur on an individual method or on a whole class, but properly decoupling unrelated objects prevents an optimization from "spilling over" into other code. Caching strategies can be hidden within the implementation of an object, and when the general codebase uses such an object and its methods often, optimizations to that object have wide applicability. Object-oriented design also provides the ability to view operations at different granularities to select the appropriate level to optimize. >>

This often requires restructuring some of the codebase. Cut and pasted code must be reabsorbed into a single method and reused. Functionality implemented slightly differently in different portions of the codebase must be factored into generally useful objects and then used everywhere. These all serve to increase the effect of any individual optimization as well as increasing maintainability and often testability.

Using a profiler to find regions of a program that run slowly is also critical to solving performance issues. This allows application of Amdahl's law by discovering those regions that take the most time first.

In the codebase for BBj Services, the code that makes object-oriented programming available to BBj developers is itself object-oriented. Previous releases of BBj had already addressed the easy performance gains, and profiling did not show any area that was significantly slower than any other. Applying some of the design principles above and refactoring the code to co-locate more of the implementation allowed the code profiler to show the critical obstacles to better performance in BBj.

This led to a number of proposed enhancements, and as the results show, better performance for BBj developers.

Results

Custom Object method calls saw the greatest improvement, but all method calls, both Java and Custom Object, are now faster than in previous releases. Overloaded parameter types and the number of methods that can match a given call site affect the optimization, but the most modest improvements are still 30% faster in BBj 9.0 than in BBj 8.31. In the best case, the time to perform a particular method call decreased by a factor of 134! The performance improvements are even more noticeable when the Custom Object definitions span several files. Notably, the time to invoke a method is now virtually the same as the time to perform a CALL when both invocations have the same number of parameters. Even the venerable CALL verb saw a small boost in performance, about 4%, in BBj 9.0.

There are even greater benefits available by leveraging some of the existing language tools. BBj 6.0 introduced the DECLARE verb to assign a type to a particular variable. The DECLARE verb instructs BBj to enforce static type checking rules on expressions involving variables used in DECLARE statement. In that release, DECLARE allowed for BBjCpl to warn about type checking errors via the -t command line option, and for the BASIS IDE to provide code completion on variables with DECLARE statements. BBj 9.0 further improves the benefit of using DECLARE statements. Method calls that only use variables specified by DECLARE statements (or expressions on those variables) can reach 90% improvement in execution time. Even the simplest method invocations are up to 10% faster on average than the same method call that includes an expression with an undeclared variable. DECLARE statements also allow BBj to infer guarantees about the object code in BBj programs that may lead to even further enhancements in the future.

Summary

Object-oriented code has the benefit of reorganizing a program to highlight similar pieces of code. This leads to refactoring opportunities, which reduces the number of lines of code. The fewer lines of code there are, the easier it is to find performance bottlenecks, and it is often easier to optimize them because of better organization. Using version 9.0, BBj developers benefit indirectly from the use of these principles, and directly, by writing decoupled, well-factored, object-oriented Business BASIC code themselves. ■