# Going *Fast, Faster, Fastest*

BBj® 12 includes significant optimizations to the string handling code and to various verbs and functions. Java 7 also includes some enhancements to the operation of the Just-In-Time (JIT) Compiler. Often, a given optimization will only apply to a specific portion of the code, or a particular programming idiom, but the optimizations included in the release of BBj 12 and Java 7 are unique in that they apply to virtually every program that runs on the BBj platform and on the Java platform.

Consider that in BBx®, string values and string variables are used on nearly every line of code. If we improve string performance even just a small amount, it will improve the performance of a large number of lines of code in a BBx program. Similarly, if we improve the performance of a large number of functions in the language, every line of code that uses a function has the potential to see some improvement. Optimizations to these core language features enhance the performance of every program written in BBx.

The release of Java 7 brings similar improvements. The Java Virtual Machine (JVM), on which the BBj platform runs, has a runtime component called a JIT Compiler. The purpose of the JIT compiler is to transform code from a platform-independent interpreted format into instructions specifically tailored to the underlying hardware. Improvements to the JIT compiler affect all Java-based programs, including BBj. This article will explain some of the optimizations and show some of the results of the efforts BASIS took to make your code run faster.

## Java 7 Improvements

Research into the operation of typical Java programs has shown that the vast majority of time executing code is spent in a small fraction of the entire codebase, and BBj is no exception. The code that the Java Virtual Machine exercises is partly dependent on BBx code interpreted by the BBj interpreter, but BBj performs common operations on every line of BBx code. As the JIT compiler compiles those operations into native code for the host platform, those operations BBj executes most frequently benefit from highly optimized code specifically targeted for the host OS and hardware.

Since BBj Services usually runs for a long period of time, often running many thousands of interpreter instances, the JIT compiler has time to continue to optimize even those portions of code that are not executed as frequently. Over time, the JVM adapts to the characteristics of your programs and tailors its optimized, generated native code to run your program as fast as possible, identifying the most likely code paths taken by the BBj codebase when running your code and ensuring those paths run as quickly as possible.

In the late versions of Oracle's release of Java 6 and with improved performance in Java 7, the JIT compiler added an important optimization called "escape analysis."

***By Adam Hawthorne***
*Software Engineer*

Escape analysis allows the JVM to eliminate unnecessary synchronization and allocation. If the JVM can determine a hard upper bound on the lifetime of a particular object, it has the opportunity to avoid allocating the object at all. With fewer constraints, it can also guarantee the object is never shared between two distinct threads of execution. That guarantee allows the JVM to eliminate any locking done on an object. All programs benefit from avoiding unnecessary work, BBj included, and since BBj often allocates objects that are never shared between two threads, eliminating unnecessary locking can produce visible gains in your BBx application.

## BBj 12 Improvements

Of course, these improvements only indirectly affect BBx code by improving the performance of the BASIS codebase. At the same time as the engineers at Oracle have been improving the performance of Java code, BASIS engineers have been busy at work directly improving the performance of BBx code. Here is some insight into what goes on behind the scenes as we improve our codebase, and ultimately, the performance of your application code.

### Soliciting Samples

In the summer of 2011, we sent a request to the developer community for samples of code that showed opportunities for performance improvements. In addition, we identified a few specific areas through our customer support transactions that we wanted to address with these changes in the string handling code. We handcrafted several tests that executed very specific individual operations ranging from common functions and verbs to various operators. We also wrote a program generator that could automatically generate programs that select from a wide variety of normal string-related

operations, including familiar ones like substring and concatenation. All these samples allowed us to have baselines from a wide array of sources so we could compare with previous versions of BBj to ensure we did not introduce any significant performance regressions with our improvements. However, in order to perform a proper comparison between the old code and the new code, we needed a test harness that could paint an accurate picture of the performance of these samples.

**Performance Testing Framework**

When getting a sense for how fast something is, it is often tempting to write a quick benchmark, run it a few times, and calculate a simple average using the arithmetic mean. Repeating this a number of times, the inaccuracy and noise evident in that kind of data becomes readily apparent (especially when trying to reproduce results to show-off the performance improvements to the management team!). BASIS' continued reliance on the cloud for its vast computational resources makes it easy to run several tests, but attempting to produce reproducible results in an environment subject to such a high degree of external influence proved a challenging task in and of itself.

*An Insider Look at BASIS Testing* at links.basis.com/12testing discusses a client/server based testing tool developed specifically for running BBj programs simply known as "the testbed." After making a few improvements to this internal tool, we adapted our sample programs to run in that environment and to execute several hundred times each, for each run of the performance test suite. We use a statistical method called bootstrapping to identify a range of median values and produce a reliable confidence interval for the median running time of each test, which allows us to determine to a reasonable degree of certainty both that our timing values are consistent and whether our results show a statistically significant increase (or decrease) in performance.

Finally, certain external effects such as the JIT compiler and the garbage collector tend to skew later numbers in the tests, so we modified the performance test suite to perform a dry run through all the tests to allow those effects to settle before finally recording the results of our tests.

**Making it Faster**

Having a way to measure the performance is certainly an important part of our optimization efforts, but actually improving the performance of the code requires its own effort. Part of that is finding where we can improve the code.

Let's look at an example that illustrates one of the motivating issues we found. To avoid the overhead of copying bytes with every assignment, BBj string values will share a buffer with other strings when possible. This sharing occurs in several ways, but by instrumenting our internal buffer handling code, while running some of our test cases, we discovered some opportunities to enhance our buffer sharing implementation. In the following figures, each group of cells represents a buffer of bytes that multiple BBj string values refer to. The shaded portion of each cell represents the portion of the buffer used by the corresponding shaded expression in the associated program text.

Originally, a typical program might have code that looks like this:

```
1: a$ = "xxxxxxxx"
2: b$ = a$
3: c$ = b$(6,3) + "xx"
```

After line 3, the buffer's contents appear as in **Figure 1**.

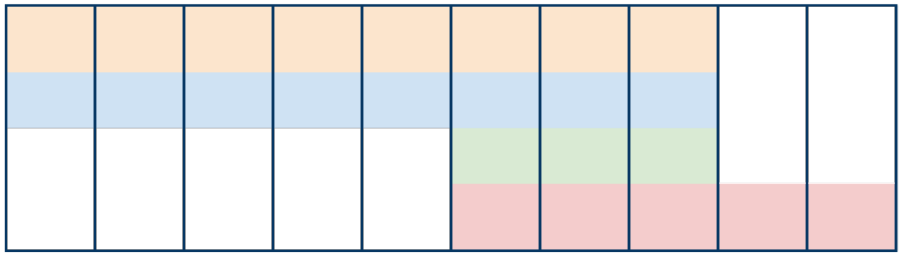


**Figure 1.** Buffer contents after line 3

```
4: b$ = c$
```

After line 4, our buffer sharing code would ensure that a$, b$ and c$ were using the same buffer at the end of the code which avoids having to copy the buffer (**Figure 2**).

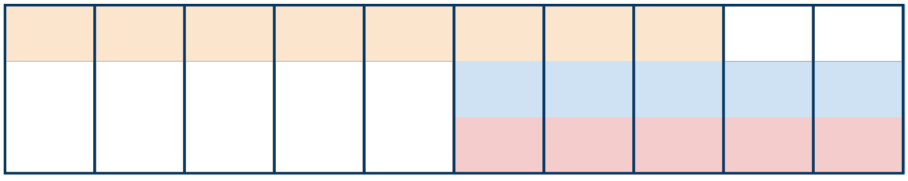


**Figure 2.** Buffer contents after line 4 (before optimizations)

The optimization opportunity presents itself when there is not enough room in the buffer in line 3 above to append the value "xx". In this case, in order to continue to be able to share the buffer, the original buffer handling code would copy the entire contents of the buffer above to a new larger buffer, copy in the contents of the appended string, and modify a$, b$, and c$ to refer to the new byte buffer. Since some copying must take place to allocate the new buffer, it is not necessary for the new buffer to continue to share the contents with the old buffer. After making this change, the buffers look like **Figure 3**.

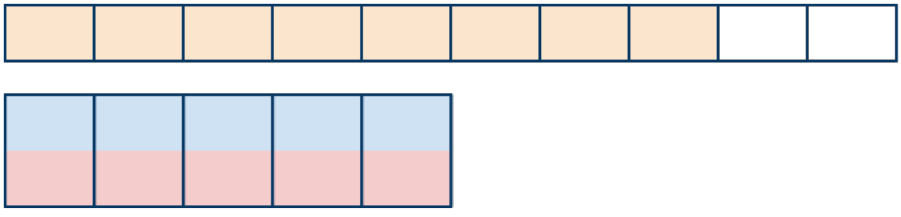


**Figure 3.** Buffer contents after line 4 (after optimizations)

Another benefit arose because of this change. When there are no longer any references to a$, we no longer need to maintain the memory allocated for that buffer and we can eliminate it. This initially requires a little more memory to maintain two buffers, but eventually, it pays off because of the ability to reclaim older buffers.

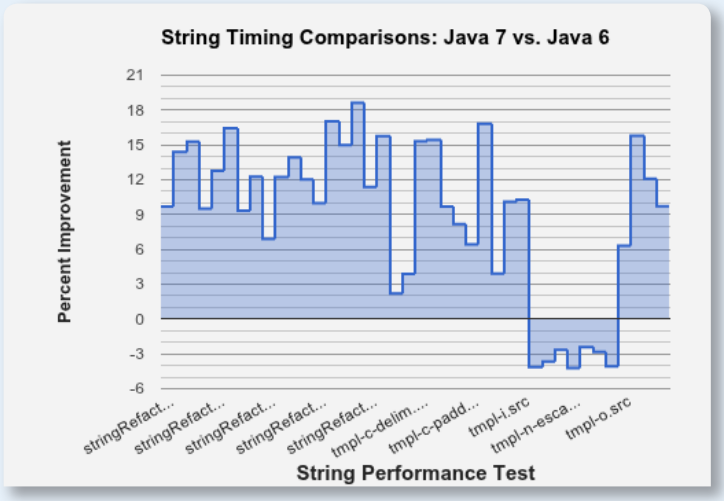
In a certain code pattern, this optimization was extremely effective because we performed this kind of operation in a loop. In this case, the beginning of the buffer (the portion referred to by a$) would continue to grow. The buffer management code cannot track the extent of each use of a buffer without suffering an extreme performance penalty, and so the act of copying the buffer in order to append would copy hundreds or even thousands of bytes that were no longer in use.

**Papercut Optimizations**

We also discovered other optimization opportunities. Many of these optimizations are only noticeable on a very small scale, but we found and implemented over a thousand of these optimizations throughout the codebase. We termed them "papercut" optimizations because individually they are not very serious, but when considered all together, they can produce a significant improvement in a program's execution time. We were able to eliminate unnecessary temporary buffers in several hundred locations throughout the BBj codebase. We tightened up certain BBx function implementations to eliminate unnecessary allocation and we improved the algorithms behind some of the common BBx functions and verbs.

## Results

The following charts are some results from our performance testing. Each bar in the graph is a specific performance benchmark as described previously.



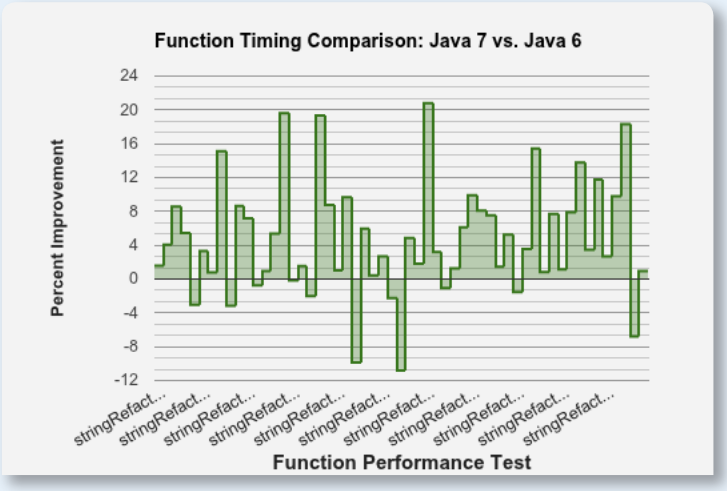




**Figure 4.** Performance comparisons in Java 7 vs Java 6

**Figure 4** shows the differences between BBj 12 on Java 7 and BBj 12 on Java 6. For the vast majority of our tests, the JIT compiler improvements in Java 7 significantly enhance performance even after our optimizations. There are a few performance regressions in Java 7, but it is important to note that these are micro-benchmarks, and each individual test is unlikely to show a significant difference in a full application. These charts together show that the overall improvements are very likely to outweigh any individual regressions.

**Figure 5** showcases the effects of our optimization efforts. It is important to note these are micro-benchmarks that exercise a specific area in our code and also verify that our optimizations are actually improving performance for those instances. For example, one can see the extreme case outlined in the above section when looking at the first bar of the first chart below, showing a dramatic 99.45% improvement in execution time (i.e., almost all the execution time disappeared). However, the combined picture of these charts again reveal an overall improvement in BBj performance, which will, in turn, improve the performance of your application code. Efforts to address the few minor regressions in these graphs are already under way.

## Summary

BASIS engineers commit themselves to improving the performance of BBx code. Often, this goes on as we do our day-to-day work, but from time to time, we take opportunities to make performance the sole focus of our efforts and the results are telling. As we continue to look for ways to optimize the BBj platform, we expect the performance of BBx programs will continue to improve so you can concentrate on just writing application code. ■

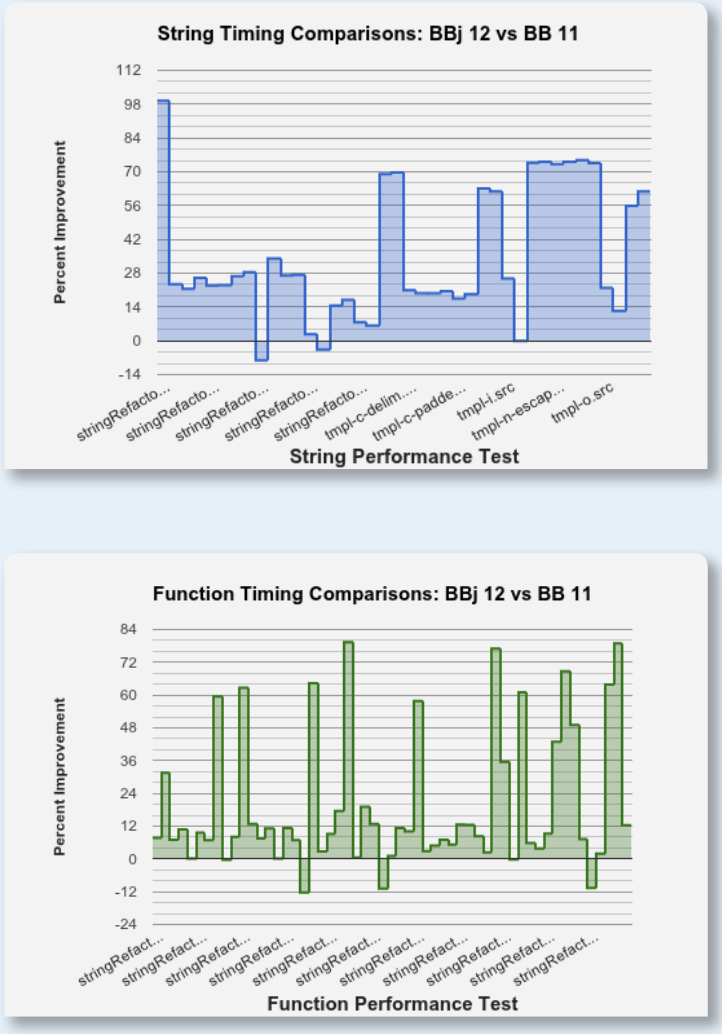



**Figure 5.** Performance comparisons in BBj 12 vs BBj 11

Find out more about the JIT compiler optimizations, both in general and specifically in Java 7
- The Java HotSpot Performance Engine Architecture White Paper at bit.ly/i1YnXn
- Java HotSpot Virtual Machine Performance Enhancements at bit.ly/L9GjBz