An Empirical Analysis of the Java Collections Framework Versus the C++ Standard Template Library

Sean Wentworth: sean.wentworth@minolta-qms.com
David Langan: langan @cis.usouthal.edu
Thomas Hain: hain@cis.usouthal.edu

School of Computer and Information Sciences
University of South Alabama
Mobile, AL  36688

Abstract

The choice of a programming language involves a tradeoff between run-time performance and factors that are more important during implementation and maintenance. In the case of two specific object-oriented languages, Java and C++, performance has favored C++. On the other hand, design-time factors such as platform-independence, flexibility, safety, and security are characteristics of Java that have made it increasingly popular. This paper presents an empirical analysis of the operation and application-level performance of two libraries supporting these languages, the Java Collections Framework and the C++ Standard Template Library. The performance of the libraries is also taken to be indicative of the performance of their respective languages, and a goal of this study is to aid in the evaluation of the above-mentioned tradeoff. The results found are consistent with other Java and C++ performance comparisons found in the literature. However, compiling Java to native machine code narrows the performance gap between C++ and Java, signaling that this may be Java's best hope for performance gains.

Keywords: Performance benchmarks, STL performance, CF performance, Java versus C++.

1 Introduction

The developer’s choice of a programming language, after other constraints have narrowed the field, involves a tradeoff between run-time performance and design-time factors that impact implementation and maintenance. In the case of two specific object-oriented languages, Java and C++, performance has favored C++. On the other hand, design-time factors such as platform-independence, flexibility, safety, and security are characteristics of Java that have made it increasingly popular over the last seven years.

Sun Microsystems’ “write once, run anywhere” marketing line for Java quickly caught developers’ attention. Java’s early evolution coincided with the rapid growth of
the Internet, which was the first proving ground for Java’s claims of platform independence. Although platform independence may have been the initial hook that lured programmers to Java, it does not account for the sustained interest. Java’s widespread acceptance comes from its flexibility, simplicity, and safety relative to other object-oriented programming languages [10]. As Tyma states, “[Java] takes the best ideas developed over the past 10 years and incorporates them into one powerful and highly supported language” [24].

Java’s rich set of packages has enticed many programmers to select the language for its handling of run-time exceptions, its support for writing network code and its support for creating multithreaded applications. Not only does Java simplify the design process, but it also ensures a high degree of safety that is not found in languages like C++. Optional thread synchronization, for example, is guaranteed and managed by the language and array bounds checking is automatically done at run-time.

While borrowing much from C++, Java deliberately omits many of the error-prone features that arise from explicit memory management in C++ [9]. Java’s memory management model provides both programmer convenience and run-time safety. The Java Run-time Environment (JRE) includes an asynchronous “garbage collector” that is responsible for deallocating heap memory thus relieving the programmer of concerns about memory leaks [12] and dangling references.

It has been shown that Java is a highly productive programming environment, generating less than half the bugs per line of code than programming in C++ [16]. However, the features that make this possible in Java come at the cost of performance. For example, the run-time performance cost of worry-free garbage collection can be substantial, especially for applications with a high rate of object creation [8].

Initially, performance degradation was not a big concern, since Java was found suitable for Internet computing tasks such as writing applets, where bandwidth and platform independence are more important than program performance. More recently, there has been increased use of Java for performance-sensitive applications, such as server applications and scientific computing, which have typically been written in languages like C++ [11]. The October 2001 issue of the Communications of the ACM
documents several ongoing projects aimed at the use of Java in the domain of high-performance computing.

This paper presents an empirical analysis of the operation and application-level performance of two libraries supporting these languages, the Java Collections Framework and the Standard Template Library. The performance of the libraries is also taken to be indicative of the performance of their respective languages, and a goal of this study is to aid in the evaluation of the tradeoff between run-time performance and design-time factors. Section 2 will outline the technological history of Java performance enhancements, and outline the characteristics of the libraries used in this study. Section 3 reviews other performance comparison studies that have been conducted. Section 4 presents the approach for performance measurement and the results of these tests. Conclusions are drawn in Section 5.

2 Background

Java is a young language and its rapid evolution is motivated by its growing popularity. Its previous performance disadvantage has been documented by numerous studies comparing the performance of Java against C/C++, however these suffer from aiming at a moving target. Section 2.1 outlines this moving target. Section 2.2 characterizes the libraries that are used in this study.

2.1 The Technological Evolution of Java

Java programs are compiled into class files composed of Java bytecode. The Java virtual machine (JVM) is a process running on a particular physical machine that interprets and executes the bytecode instructions. Initially JVMs were purely interpreted. Later, the interpreted model was improved with just-in-time (JIT) compilers that compiled Java bytecode to native machine code at method load-time. Depending on the application, a JIT-enabled JVM can boost execution performance from a factor of 2 to 20 [10][12][26]. Building on the JIT concept, Sun improved Java’s performance further in its HotSpot Performance Engine, which incorporated several performance enhancing technologies [23]. In both JIT and HotSpot, optimizations are performed during program execution. To avoid this run-time overhead, native-code compilers such as Excelsior's
JET [5] were developed for Java. The Marmot optimizing compiler for Java has yielded "application performance …approaching that of C++" [6].

As the performance gap between Java and natively compiled languages has narrowed, many companies are promoting Java as a general-purpose high level computing language [1]. There is an emerging trend to use Java in place of C++ to implement large-scale applications [3]. In 1997, it was estimated that as many as 60% of C++ developers were learning Java [7]. In addition, many computer science curriculums have replaced C++ with Java as the language used to teach introductory programming concepts [24]. This growing language shift has created the need for a better understanding of Java's performance relative to more traditional object-oriented languages, particularly C++. In the literature, Java is most often compared to C++ [12]. This is probably due to their syntactical and semantic similarities, and because C++ has become the de facto standard among general-purpose, object-oriented languages.

2.2 The Libraries

What follows is a brief overview of the libraries that were compared: the Java Collections Framework (CF) and the C++ Standard template Library (STL).

2.2.1 The Standard Template Library (STL)

The STL became part of the ANSI/ISO standard definition of the C++ language in 1994. It is organized around three fundamental components: containers, algorithms and iterators. Supporting these fundamental components are three additional components: allocators, adaptors and function objects.

As the name implies, containers are objects that hold other objects. The seven containers provided in the STL are grouped into sequence containers (vector, list, deque) and associative containers (set, multiset, map, multimap). Interestingly, containers are not accessed via references (or pointers) as might be expected in C++, but rather using copy semantics, thereby providing both efficiency and type safety [15]. In addition, containers free the programmer from allocation and deallocation concerns by generally controlling these aspects using default allocators [21].
Algorithms perform the manipulation of elements in containers. However, algorithms are decoupled from container implementations in a completely orthogonal way by the use of iterators, thus providing genericity. Algorithm templates are loosely grouped into seven functional categories: non-mutating sequence algorithms, mutating sequence algorithms, sorting and searching algorithms, set operations, heap operations, numeric algorithms and miscellaneous/other functions [15].

2.2.2 The Java Collections Framework (CF)

Sun’s initial releases of the SDK included very limited support for data collections and algorithms (only the vector, stack, hashtable and bitset classes). It was not until the release of SDK 1.2 in 1998, that Sun introduced an extended generic container library called the Collections Framework (CF). The primary conceptual difference between the CF and the STL is that the former is focused on containers rather than on the combination of containers and algorithms [25]. The STL defines both containers and algorithms as relatively equal entities that can be mixed and matched in useful ways. The CF, on the other hand, defines algorithms as methods of either the collection interface or class. Because of this singular focus, the CF does not fully support generic programming as defined by Musser and Stepanov [14] since the algorithms are tightly coupled with the containers on which they operate. The CF defines only a small number of algorithms that perform basic operations such as sorting, searching, filling, copying and removing, while the STL offers these and many additional ones such as heap operations, numeric algorithms, find operations, and transformations.

The CF containers are defined in a hierarchy of Java interfaces. Interestingly, the interfaces do not derive from a single "super-interface," since the CF does not treat maps as true collections of objects as does the STL. The CF designers decided to differentiate between a Collection, as a group of related objects, and a Map, as a mapping of key/value pairs. However, the map interfaces have "collection view" operations, or wrappers, that allow them to be manipulated as collections. That is, a Map can return a Set of its keys, a Collection of its values or a Set of its pairs [4]. The CF also supplies six concrete classes based upon these interfaces (shown in parentheses with its associated interface in Figure
1). In addition, an abstract class definition of each of the interfaces is provided, giving programmers a partial implementation from which custom implementations can be built.

---

![Collections Framework interface hierarchy with concrete implementations in parentheses](image)

**Figure 1** Collections Framework interface hierarchy with concrete implementations in parentheses

### 3 Previous Performance Comparisons

Previous performance comparisons between Java and C or C++ fall into three general categories. The languages are compared using full-scale applications, smaller application kernels, and low-level microbenchmarks. Although they provide mostly anecdotal data, the full-scale application comparisons offer valuable insight into the language differences on real world applications. Application kernels test a language's ability to perform certain general types of operations, such as the calculation of linear equations. Microbenchmarks are used to analyze specific low-level language operations, such as method calling, variable assignment, or integer addition. Interestingly, the majority of these studies show similar results: Java code running on a JIT-enabled JVM requires about 3 times the run-time of corresponding C or C++ code.

#### 3.1 Full-Scale Applications

Jain et al. designed an image processing application called MedJava and compared it to xv, an equivalent image processing application written in C [10]. Eight image filtering operations, each applied to images of various sizes, were timed with each application. Although MedJava met or exceeded the performance of xv in two of the filters, for the remaining six filters, xv executed from 1.5 to 5 times faster than MedJava. This rather large range can be attributed to the designs of the eight image filtering operations. For example, the authors point out that the Java version of the sharpen filter
performs better than the xv version because this filter is dominated by floating point to integer conversions, which are less efficient in C/C++. A second comparison was performed using transport interface applications over a high speed ATM network to transfer image files. This comparison showed C++ outperforming Java by only 15-20% over a range of sending-buffer sizes.

Another large-scale comparison between Java and C++ analyzed a parallel multipass image rendering system in both languages [26]. Based on the same algorithm, the two versions were tested on three images using machines with a varying number of CPUs. The C++ version executed 3 to 5 times faster than the Java version. The I/O capabilities of the two versions were also examined by measuring the elapsed time for each to read the modeling data into memory. The C++ version read the data about 5 times faster than the Java version for the three images.

Prechelt compared Java to C and C++ by having 38 different programmers write 40 implementations (24 written in Java, 11 in C++, and 5 in C) of a program that converts a telephone number into a word string based upon a dictionary file [17]. Similar to the previous two studies, his results showed that the median execution time of the C and C++ codes were just over 3 times faster than the Java implementations.

3.2 Application Kernels

Bernardin compared Java to both C and C++ using application kernels that multiply univariate polynomials of varying degrees [1]. In the Java to C comparison, the polynomials were represented as arrays of 32-bit integers. The C version was compiled two ways: with standard compilation options, and with powerful, machine-specific optimizations. The heavily optimized C code executed 2.3 times faster than the Java code. However, against the standard C compilation, the Java version performed better, averaging 21% faster. The Java to C++ comparison represented the polynomials as arrays of pointers to generic coefficient objects. Using this design, the C++ version executed an average of 2.6 times faster than the Java version.

The Linpack benchmark—a standard application kernel benchmark—was performed in both Java and C using a problem size of 1000×1000 [3]. This benchmark is a matrix multiplication of linear equations used to measure floating-point numeric
performance. The results, measured in floating point operations per second, showed C running about 2.3 times faster than Java.

3.3 Microbenchmarks

In Roulo's analysis, several microbenchmarks were performed comparing Java to both C and C++ [18]. He showed that across methods accepting from 1 to 11 parameters, Java’s method calls are approximately one clock cycle longer than the same C++ function calls. He also showed that explicit object creation in Java is roughly equivalent to using the `malloc` operation in C or creating objects with the new operator in C++. This is due to the fact that in all three of these instances, the memory allocation occurs on the heap. However, such user-defined objects are not the only objects that Java and C++ must create and manage. Many common operations require the creation of implicit, temporary objects. Java uses the heap for all object creation, whereas C++ creates temporary objects on the system stack, which “is almost always in the on-board Level 1 cache” [18]. Therefore, temporary object creation in C++ occurs about 10 to 12 times faster than it does in Java. Furthermore, the use of JIT seems to have little impact on the Java performance penalty for object creation [18].

Mangione also used microbenchmarks to compare Java and C++ [12]. Several fundamental computational tasks, such as method calls, integer division and float division, were performed in loops that executed 10 million times. In six of the loops, the Java versions executed as fast as the C++ versions. At first glance, these results seem hard to believe, given the other timings presented so far. There is good reason for skepticism here. Such simple looping microbenchmarks can yield widely varying results between runs due to the behavior of cache memory and the size of internal write buffers [2]. Even when such factors are taken into account, conclusions drawn solely from microbenchmark results should be carefully qualified.

4 Performance Comparisons

This paper presents a comparison of the support offered to Java and C++ for data collections. More specifically, it compares the performance of the CF and the STL. The performance tests are done at two levels: a microbenchmark compares equivalent
algorithm/method operations on the various containers, and a combination benchmark compares overall performance given various execution profiles. This two-tiered approach to performance testing is modeled after the methodology used in several other empirical performance comparisons [10] [19] [20].

4.1 Microbenchmarks

4.1.1 Comparable Library Features

As described in Section 2.2, the approaches used by the STL and the CF differ in several significant ways. In order to compare operations on containers of each library, it is necessary to identify the overlap in their functionality, since this will provide the basis for the microbenchmark tests conducted.

For each library, the functionality of the implemented container classes was compared. This analysis yielded six functionally comparable container classes (highlighted in Table 1). Since the CF does not contain a queue container, and because queues are nonetheless commonly used data structures, a queue class was created by extending the LinkedList class and adding enqueue and dequeue operations.

For the set and map container types, the CF provides two possible container classes—an "ordered" and a "hashed" version. Currently, the STL only provides an ordered set implementation. Because its functionality more closely matches that of the STL, the ordered set of the CF (TreeSet) was chosen for comparison against the STL's set container. Non-standard hashed set templates are available for C++ in libraries such as STLport [22]. It is expected that the next version of the STL will contain a hashed set implementation [13].
The initial analysis of algorithms/methods available for these six containers yielded a set of comparable algorithms/methods across both libraries that represented approximately 47% of the total. However, not all of the comparable methods were included in the final benchmark design. Some were deemed computationally insignificant. For example, the method to obtain a container's size simply returns the value of the container's privately stored size counter. Some methods were not included because they were not functionally equivalent in both libraries. For example, the CF ArrayList constructor, ArrayList(int n), creates an empty ArrayList container and pre-allocates room for n elements. Although space has been allocated, the container is empty after the constructor is called (the size() method returns 0). The syntactically comparable constructor in the STL, however, fills the container with n constructed elements in addition to allocating space (so size() returns n). Although the constructors appear to be functionally equivalent, they are not. Adding an element to the front of the CF’s ArrayList after this constructor is called is much faster than in the STL version, because the STL must shift all n elements currently in the vector before adding the new element.

After removing such methods from the initial list, the set of usable methods represented about 35% of the total available methods. If both a member method and a

---

**Table 1: Microbenchmark containers and timed operations.**

<table>
<thead>
<tr>
<th>Tests Performed</th>
<th>Dynamic Array</th>
<th>List</th>
<th>Ordered Set</th>
<th>Ordered Map</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create &amp; Fill</td>
<td>Create &amp; Fill</td>
<td>Create &amp; Fill</td>
<td>Create &amp; Fill</td>
<td>Equal</td>
<td>Create &amp; Fill</td>
<td></td>
</tr>
<tr>
<td>Copy</td>
<td>Sequential Search</td>
<td>Equal, min/max</td>
<td>Equal, min/max</td>
<td>Push &amp; Pop</td>
<td>Equal</td>
<td></td>
</tr>
<tr>
<td>Sequential Search</td>
<td>Sort</td>
<td>Increment values of each element</td>
<td>Increment values of each element</td>
<td>Enqueue &amp; Dequeue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>Binary Search</td>
<td>Subset relationship</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search</td>
<td>Reverse, equal, min/max</td>
<td>Set Union</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shuffle, reverse, equal, min/max</td>
<td>Increment values of each element</td>
<td>Set Intersection &amp; Set Difference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increment values of each element</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
generic algorithm were available for a particular operation, the member method was chosen for the benchmark since they are generally more efficient than generic algorithms by having access to a container's internal structure.

The tested containers were filled with objects consisting of an integer and two floating point numbers. These data elements were initialized with random values and were subsequently used in “operations” done to each item in a container. Each test begins by creating and filling either four or five containers with 200,000 elements each and recording the execution time for this process. Additional tests that vary by container are performed and the execution time for each test is recorded. Table 1 identifies the containers and timed tests that make up the microbenchmark portion of the suite.

There are two advantages to designing the microbenchmarks in the suite around common library operations. First, using such a high level of granularity helps to ensure that the suite is robust. The difficulty with many microbenchmarks comes from the fact that they typically measure performance on the order of source-code-level instructions (e.g., arithmetic operations or assignment). Therefore, results can be highly susceptible to variation due to architectural implementations such as cache performance and read buffers [2], as well as to compiler optimizations. Any of these factors may result in microbenchmark performance that does not correlate with performance in actual applications. Building microbenchmarks from high-level library operations avoids such threats to the validity of the results. Second, the microbenchmark results serve a dual purpose. In addition to being merely interpretive tools for larger scale testing, the microbenchmark results provide valuable data in their own right. Being aware of the performance ratios for specific library operations is helpful to potential users of the libraries, especially if one is considering porting code from one language to the other.

Measures were taken to minimize variability among the Java microbenchmarks. The System.gc() method was called before each timed test to attempt to force garbage collection to take place outside of the timings. This minimized the possibility that garbage collection would occur in the middle of a test yielding misleading results. Since garbage collection is an inherent part of Java, one might argue that attempting to exclude it from the microbenchmark timings invalidates the comparison between the CF and the STL. This is not the case. The attempt to control garbage collection is only done in the
microbenchmark part of the suite. No such adjustments were made in the combination benchmark. Also, microbenchmarks are intended to measure the performance of specific, limited operations. Garbage collection occurs "as needed" at unpredictable times during the execution of a Java program. If a garbage collection cycle occurred during a microbenchmark test, it would likely be reclaiming space from some previous test in the microbenchmark. This would result in the garbage collection overhead being attributed to the wrong test. Thus, attempting to keep garbage collection out of the timings helps to insure the accuracy of the microbenchmark results.

4.1.2 Microbenchmark Results

Figure 2 shows the microbenchmark performance ratios for the CF with HotSpot enabled and the compiled CF versus the STL (normalized to 1). The majority (22) of the CF with HotSpot to STL ratios fall in the range from 1.0 to 7.7. Three of the ratios, list binary search, set union and set intersection & difference, are extremely outside of this range due to the results from these STL microbenchmark operations (see Table 2 for a complete listing of microbenchmark ratios).

The STL set union test and set intersection & difference test ran exceedingly long (nearly 40 times longer than the Java versions). The reason for this lies in the differences between how these operations are performed in the CF and the STL. The CF performs these operations and merges the results into one of the original sets. For example, performing set union on set1 and set2 results in set2 being unchanged and set1 now being the union of set1 and set2. The STL, on the other hand, performs these operations by creating a third container to receive the result of the operation. Since several of these operations are performed in the microbenchmark, many new sets are created. The memory requirements for these containers quickly exceeds the 128 Mb of physical memory on the test machine, which results in a good deal of memory swapping to the disk (noticed during the execution of these tests). Interestingly, compiling this "problematic" STL code with Borland's compiler yielded results that were 25 times faster than Microsoft's for set union and 15 times faster for set intersection & difference.
Figure 2  Graph of microbenchmark results for the CF with HotSpot enabled, the CF compiled and the STL.
Table 2: Microbenchmark performance ratios.

<table>
<thead>
<tr>
<th>Containers and Operations</th>
<th>Java to Java Ratios</th>
<th>Java to C++ Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interpreted vs Hotspot</td>
<td>Hotspot vs compiled</td>
</tr>
<tr>
<td>D Y N A M I C   A R R A Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Create, fill</td>
<td>3.25</td>
<td>1.80</td>
</tr>
<tr>
<td>2 Copy</td>
<td>4.97</td>
<td>1.20</td>
</tr>
<tr>
<td>3 Sequential Search</td>
<td>6.12</td>
<td>0.96</td>
</tr>
<tr>
<td>4 Sort</td>
<td>4.04</td>
<td>1.38</td>
</tr>
<tr>
<td>5 Binary Search</td>
<td>5.03</td>
<td>1.04</td>
</tr>
<tr>
<td>6 Shuffle, reverse, equal, min/max</td>
<td>4.45</td>
<td>1.13</td>
</tr>
<tr>
<td>7 Reset all values</td>
<td>6.13</td>
<td>1.34</td>
</tr>
<tr>
<td>L I S T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 Create, fill, copy</td>
<td>1.48</td>
<td>2.93</td>
</tr>
<tr>
<td>9 Sequential Search</td>
<td>4.20</td>
<td>1.04</td>
</tr>
<tr>
<td>10 Sort</td>
<td>4.25</td>
<td>1.42</td>
</tr>
<tr>
<td>11 Shuffle, reverse, equal, min/max</td>
<td>4.34</td>
<td>1.43</td>
</tr>
<tr>
<td>12 Reset all values</td>
<td>4.75</td>
<td>1.34</td>
</tr>
<tr>
<td>S E T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 Create, fill</td>
<td>1.68</td>
<td>2.57</td>
</tr>
<tr>
<td>14 Equal, min/max</td>
<td>6.03</td>
<td>1.43</td>
</tr>
<tr>
<td>15 Reset all values</td>
<td>4.49</td>
<td>1.53</td>
</tr>
<tr>
<td>16 Subset Relationship</td>
<td>6.82</td>
<td>1.45</td>
</tr>
<tr>
<td>17 Set Union</td>
<td>6.55</td>
<td>1.25</td>
</tr>
<tr>
<td>18 Set Intersection &amp; Difference</td>
<td>6.51</td>
<td>1.60</td>
</tr>
<tr>
<td>M A P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 Create, fill</td>
<td>1.63</td>
<td>1.94</td>
</tr>
<tr>
<td>20 Equal, min/max</td>
<td>5.13</td>
<td>1.02</td>
</tr>
<tr>
<td>21 Find, reset, remove</td>
<td>6.22</td>
<td>0.89</td>
</tr>
<tr>
<td>S T A C K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 Equal</td>
<td>5.25</td>
<td>0.77</td>
</tr>
<tr>
<td>23 Push &amp; pop</td>
<td>2.86</td>
<td>1.79</td>
</tr>
<tr>
<td>Q U E U E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 Create, fill</td>
<td>1.27</td>
<td>7.80</td>
</tr>
<tr>
<td>25 Equal</td>
<td>4.58</td>
<td>1.56</td>
</tr>
<tr>
<td>26 Enqueue &amp; Dequeue</td>
<td>1.70</td>
<td>3.40</td>
</tr>
<tr>
<td>R A T I O   A V E R A G E S</td>
<td>4.21</td>
<td>1.70</td>
</tr>
</tbody>
</table>

These averages are excluding tests 17 & 18.
In order to get a composite understanding of the microbenchmark results, the performance ratios were given equal weight and averaged together. It would have been better to weigh the operations based upon their relative frequency of use, but no data was found that supported an analysis of this type. Before calculating the average ratio, it is important to exclude the three tests described above as outliers. Including them results in an average ratio that is representative of neither the outlier ratios nor the more consistently grouped ratios. The average of the remaining 24 CF with HotSpot to STL microbenchmark ratios is 3.15. This ratio is consistent with the performance comparisons between Java and C++ found in the literature.

Compiling the CF to native machine code improved the performance of the CF microbenchmarks by an average factor of 1.70. Thus the compiled CF performed better against the STL than the CF with HotSpot. The average of the 24 compiled CF to STL microbenchmark ratios is 2.31. This is slightly better than what was seen in the literature. It is interesting to note that the CF with HotSpot versus the purely interpreted CF improved the performance of the microbenchmark results by an average factor of 4.21. It's no surprise that some form of JIT technology like HotSpot has become the norm for Java execution.

4.2 Combination Benchmark

The combination benchmark is the second part of the suite. A tool was designed to facilitate the creation of these tests. The tool provides the ability to mix and match containers and algorithms tested in the microbenchmark part of the suite to create a single combination test profile. The resulting profile can then be tested with each library.

The combination benchmark provides a more realistic simulation of containers’ specific algorithmic functionality than the microbenchmarks. This is because these tests are performed on a larger scale—analogous to application kernels. Normal program overhead, such as Java’s garbage collector, is present.

For testing purposes, four combination benchmark profiles were selected that represented a range of possible profiles. At one extreme, a profile was created that
performs every available* operation for each of the six containers an equal number of

times. At the other extreme, a profile was created that performed none of the container

operations. It simply read the data files and filled a dynamic array container with

elements and then moved these elements from one container to the next until each of the

six container types had been filled and emptied. The remaining two profiles represent

two points within the spectrum bounded by these first two profiles. They were defined by

dividing the available operations into categories borrowed from the STL specification of

mutating and non-mutating [19]. Mutating operations are operations that modify the

elements of the container in some way, either by changing the order of the elements or by

changing the values of the elements. Conversely, non-mutating operations do not modify

the elements of the container. The profile of mutating operations includes all six

containers. The profile of non-mutating operations includes only four containers,
because there are not any non-mutating operations available for the Stack and Queue

containers. The binary search operation was left out of the non-mutating profile, because

a mutating operation (sort) is required before a binary search can be performed. Across

the four profiles, container sizes were held constant and each selected operation was

performed an equal number times. Also, the order in which the containers were selected

for a profile remained constant for each of the four profiles.

4.2.1 Combination Benchmark Results

Figure 4 shows the combination benchmark performance ratios for the CF with

HotSpot enabled and the compiled CF versus the STL (normalized to 1). The results

generally reflect the overall results of the microbenchmarks. The average of four

HotSpot combination ratios is 3.15. As was the case with the microbenchmarks, this

ratio is consistent with the performance comparisons between Java and C++ found in the

literature.

The compiled CF combination benchmark was faster than the CF with HotSpot by

an average factor of 1.54. The average of the compiled CF to STL combination

* The combination benchmark was designed to operate on one container at a time, so binary operations

found in the microbenchmarks, such as equality, are not present in the combination benchmark.
benchmark ratios is 2.04. Again, compiling Java narrows the performance gap between Java and C++.

Figure 4: Graph of the combination benchmark performance ratios for the CF with HotSpot enabled and the compiled CF (the STL is normalized to 1).

4.2.2 Caveats

A few caveats need to be addressed at this point regarding the interpretation of the results reported here. An effort was made to make all tests as “equal” as possible, but there are some inherent differences in the languages that affect the flexibility and efficient use of the containers described here. For example, the decision to use objects as the type of data stored in the benchmark containers was made to ensure an "apples to apples" comparison of performance. The STL container classes can be used to hold either objects or primitive data types. The CF classes are designed only to hold objects, thus if primitive data type values need to be stored in a container the programmer would be required to perform type conversions and to use wrapper classes, incurring additional
overhead. Thus the STL template approach offers some flexibility not available in the Java approach.

The C++ code used in the benchmarking was written to be as efficient as possible, not to just “mimic” Java’s behavior. For example, in the C++ code many of the objects created were on the data stack as opposed to being created dynamically on the heap as is done in Java. While it would have been possible to force the C++ solution to mimic the Java behavior, doing so would have artificially degraded the C++ performance and would not have represented “best practice” in using C++.

Programmers generally do not need to know any of the internal details of the STL or CF implementations to use those libraries. However, to use those libraries at “maximum efficiency” such knowledge may be needed in some cases. The designers of these libraries made tradeoff decisions in the process of selecting their representation techniques. Often their choices may improve the efficiency of one operation at the expense of another. Thus in both languages an awareness of “best practice” rules is needed to achieve optimal results.

5 Summary and Conclusions

Previously reported performance comparisons showed that C++ programs tend to be faster than equivalent JIT-enabled Java programs by a factor of about three. The results presented here showed that the performance of the Collection Framework compared to the STL, based on using HotSpot, had a similar ratio. The degree of variation seen between tests in the microbenchmarks are likely to be due to differences in algorithmic implementations of the library operations and data representations.

Although the results of running Java with HotSpot matched the ratios found in other comparisons, the tests revealed that compiled Java code yielded ratios relative to C++ averaging between 2.0 and 2.5. These results are promising for those who do not require code portability, but who do desire the other advantages offered by Java. Furthermore, compared to C++ compilers, the development of static Java compilers is relatively immature. Currently, only a few static compilers are available for Java. As the concept of statically compiling Java applications becomes more popular, there are likely to be significant improvements in static Java compilation technology.
This study extends the body of knowledge regarding the performance of JIT enabled Java compared to C++ by providing a methodical comparison of the performance of their respective collection classes. It also provides a data point regarding the performance of compiled Java code to C++ code.

References


