Submission to 'Communications of the ACM'

Comparing Java vs. C/C++ efficiency differences to inter-personal differences

Lutz Prechelt (prechelt@acm.org)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/608-7343

March 23, 1999

The relative efficiency of Java programs is much discussed today, in particular in comparison to well established implementation languages such as C or C++. Java is often considered very slow and memory-intensive. Most benchmarks, however compare only a single implementation of a program in, say, C++to one implementation in Java — neglecting the possibility that alternative implementations might compare differently. In contrast, the current article presents a comparison of 40 different implementations of the same program, written by 40 different programmers. The data allows for comparing, for one particular programming task, the *average* relative performance between languages as well as the performance differences from one programmer to another within a group of programs written in the same language. As we will see, these inter-personal program differences are larger than those between the languages — and the performance gap between Java and other languages is still shrinking rapidly.

Origin of the data

The 40 program implementations investigated in this report were created by graduate students in the course of a controlled experiment on a different question [1]. There are 24 programs written in Java, 11 in C++, and 5 in C. Each program was written by a single person. These programmers had an average of 8 years of programming experience and estimated they had previously written a average of 100 KLOC each (median: 20 KLOC).

All programs implement the same functionality, namely a conversion from telephone numbers into word strings as follows. The program first loads a dictionary of 73113 words into memory from a flat text file (one word per line, 938 Kilobytes overall). Then it reads "telephone numbers" from another file, converts them one by one, and prints the results. The conversion is defined by a fixed mapping of characters to digits.

The task of the program is to find a sequence of words such that the sequence of characters in these words exactly corresponds to the sequence of digits in the phone number. All possible solutions must be found and printed. The solutions are created word-by-word and if no word from the dictionary can be inserted at some point during that process, a single digit from the phone number can appear in the result at that position. Many phone numbers have no solution

DRAFT

at all. Here is an example of the program output for the phone number “3586-75”, encoded using a German dictionary:

```
3586-75: Dali um
3586-75: Sao 6 um
3586-75: da Pik 5
```

A list of partial solutions needs to be maintained by the program while processing each number and the dictionary must be embedded in a supporting data structure (such as a 10-ary digit tree) for efficient access. Search functions of such kind might be part of a server in a larger client/server software system.

The programmers were asked to write as reliable a program as they could. Efficiency was called less important. However, a run time limit (not quantified to the programmers in advance) was imposed during the acceptance test and many programs failed to satisfy it in the first attempt and had to be optimized before they were accepted. Across all 40 programmers, writing the program took between 3 and 63 work hours (median: 10 hours, mean: 14 hours) and the resulting program had between 107 and 614 lines (median: 244 lines, mean: 277 lines, excluding comments).

All measurements presented below were taken on a Sun Ultra 1 Unix workstation with 192 MB main memory running the SunOS 5.5.1 operating system. The C/C++ programs were compiled with the GNU gcc/g++ compiler version 2.7.2, the Java programs ran under Sun’s JDK 1.2 reference implementation with just-in-time compiler (JIT).

How to analyze the data

For each metric of interest (such as the runtime of the program), we could represent the values for each language group by their mean, but doing that we would throw away a lot of interesting information. Therefore, we represent the entire distribution by so-called boxplots. Each graph contains four boxplots: one for the values obtained from the Java programs, one for the C++ programs, one for the C programs, and one for the union of the C and C++ programs. The box indicates the location and extent of the ”middle half” of the values, i.e., the left edge is positioned so that just 25% of the values are smaller (the 25% quantile), the right edge so that just 25% of the values are larger (75% quantile). The T-shaped whiskers indicate the 10% and 90% quantiles, respectively. The fat dot inside the box is the 50% quantile, usually called the median. The letter M in the plot represents the arithmetic mean of the data and the dashed line is plus/minus one standard error of the mean. For comparing the averages of two sets of data, the median is often more appropriate because it is not influenced by outliers.

The other interesting aspect of the data is its variability. Again, the most common measure, the standard deviation, is sensitive to outliers. The width of the box is a robust measure of variability. Note that the 25median of the lower half of the data and the 75box edge) is the median of the upper half of the data. Hence, the 25be called average representatives of the good (efficient) and bad (inefficient) programs, respectively. We call the quotient of these two values the ”good/bad ratio”. I will actually present the bad/good ratio instead, because that is easier to interpret.

Memory consumption differences

Let us first investigate the memory requirements of the different programs. Figure 1 shows the amount of memory required by the programs after they have loaded the dictionary and then processed 1000 telephone numbers. The memory size reported includes the size of static and dynamic data structures, the program code and libraries used, and the basic process overhead. For Java programs it also contains the size of the Java Virtual Machine, including the just-in-time compiler.

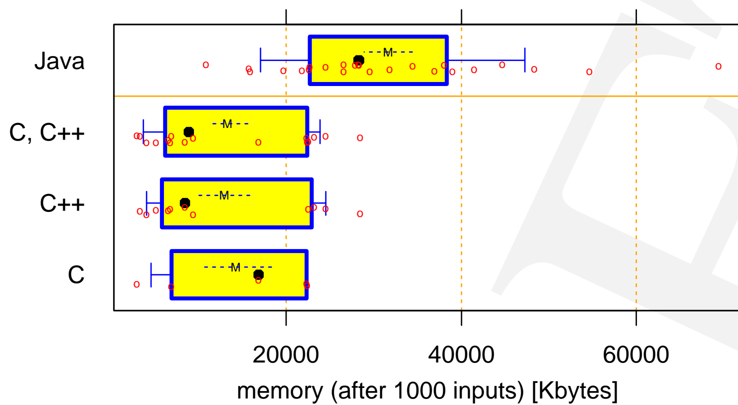

Figure 1: Total memory consumption for the various programs.

We make the following observations:

1. The average memory requirement of the Java programs is two to three times that of the C or C++ programs.
2. Even the most modest Java programs require a little more memory than the average C/C++ programs and 3 to 4 times as much as the best C/C++ programs.
3. The good/bad ratio is 3.7 for the C/C++ programs and 1.7 for the Java programs.

As wee see, the Java programs indeed require substantially more memory on average, but the ratio to the C or C++ programs is not larger than the good/bad ratio.

Run time differences

The total CPU time required by the programs (*run time*) consists of two parts; one for loading the dictionary (*load time*) and one for actually processing the 1000 inputs (*processing time*). Figure 2 shows the total run time.

These are the main observations:

1. The median run time of the Java programs is over three times that of the C/C++ programs. Due to four huge outliers in the Java group (361, 360, 151, 130 minutes, not shown in the plot), the mean is very different from the median in the Java group so that it is 18 times the mean of the C/C++ group.

   The ratio of the median load time is 6 to 1 for Java versus C/C++, the ratio of median processing time is 3 to 1.

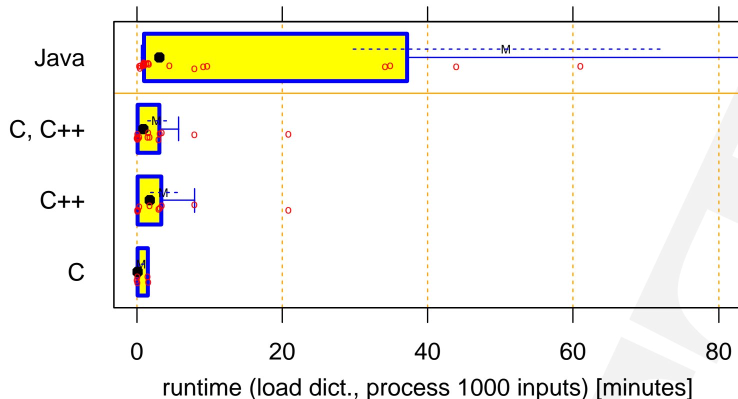

Figure 2:

2. The three fastest Java programs are about twice as fast as the median C/C++ program and 10 times slower than the three fastest C/C++ programs.

3. The run time good/bad ratio is 32 for C/C++ and 37 for Java.

   This high variability stems from rather huge differences in the actual search routines: The load time good/bad ratio is only 3 for C/C++ and 5 for Java, but the processing time good/bad ratio is as high as 153 for C/C++ and 71 for Java!

4. The C programs are substantially faster than the C++ programs. Due to the small number of C programs the ratio can not be quantified accurately.

The processing time (and hence also the total run time) reflects that part of the programming task that is not straightforward but rather requires substantial design considerations by the programmer. Promptly, the individual differences are huge compared to the differences between languages, even though the latter are quite large as well.

The memory, run time, and processing time data is summarized in Figure 3. From the error bars in that graph we can learn another important lesson: Large performance ratios like many of those presented here are unstable even if they are estimated from a substantial number of programs. Consequently, performance comparisons based on only a single program pair should be considered highly dubious unless it is guaranteed that both programs are equally well designed — appropriate for the language in which they are written.

Other differences

The above results are actually biased against the Java programs: On average, the Java programmers had only half as much programming experience in Java as the C programmers had in C or the C++ programmers had in C++. On the other hand, no clear relationship between the programming experience and the run time or memory efficiency of the resulting program could be found in the data. The work time required for writing the program is also quite similar for the Java group versus the C/C++ group, except for three Java outliers who took over 30 hours.

The length of the resulting programs (excluding comments) is similar in all three groups but the Java programmers inserted a significantly larger amount of comments into their programs than the C++ programmers and even much more than the C programmers.

DRAFT

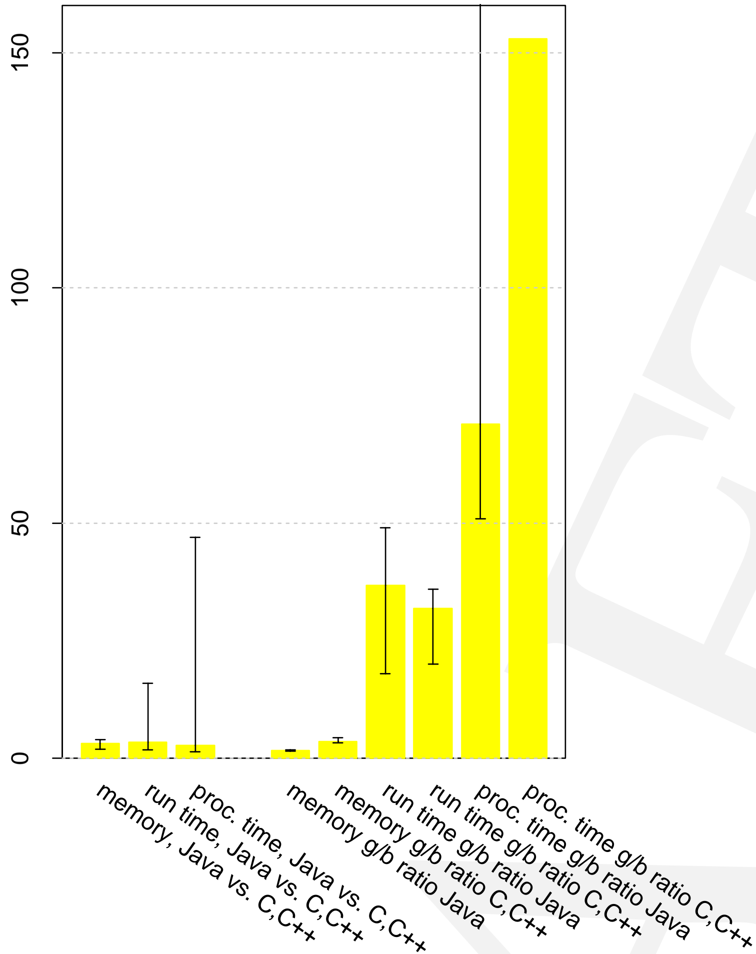

Figure 3: Left bar group: Ratios of Java group versus union of C and C++ group for median memory consumption, median total run time, and median pure processing time. Right bar group: good/bad ratios for each of the groups separately. The error bars indicate 50% confidence intervals (computed by bootstrapping).

Conclusions

I see three important conclusions from this data:

- As of JDK 1.2, Java programs are typically indeed much slower than programs written in C or C++. They also consume much more memory.
- However, even for one and the same language the inter-personal differences between implementations of the same program written by different programmers (good/bad ratio) are much larger than the average difference between Java and C/C++. Performance ratios of factor 30 or more are not uncommon between the median programs from the upper versus lower half. As a result, an efficient Java program may well be as efficient as (or even more efficient than) the average C or C++ program for the same purpose.
- As a consequence, it is wise to train programmers for writing efficient programs and/or to ensure reasonable efficiency by means of design and code inspections.

Similar conclusions probably apply to many kinds of tasks, though certainly not to all. In particular, the programming problem investigated here required a non-trivial algorithm and data structure design. However, the data clearly shows that the importance of an efficient technical infrastructure (such as language/compiler, operating system, or even hardware) is often vastly overestimated compared to the importance of a good program design and an economical programming style.

DRAFT

References

[1] Lutz Prechelt and Barbara Unger. A controlled experiment on the effects of psp training: Detailed description and evaluation. Technical Report 1/1999, Fakultät für Informatik, Universität Karlsruhe, Germany, March 1999. ftp.ira.uka.de.

DRAFT