Lisp as an Alternative to Java

Erann Gat
Jet Propulsion Laboratory, California Institute of Technology
Pasadena, CA 91109
gat@jpl.nasa.gov
9 November 1999

Introduction

In a recent study [1], Prechelt compared the relative performance of Java and C++ in terms of execution time and memory utilization. Unlike many benchmark studies, Prechelt compared multiple implementations of the same task by multiple programmers in order to control for the effects of differences in programmer skill. Prechelt concluded that, "as of JDK 1.2, Java programs are typically much slower than programs written in C or C++. They also consume much more memory."

We have repeated Prechelt's study using Lisp as the implementation language. Our results show that Lisp's performance is comparable to or better than C++ in terms of execution speed, with significantly lower variability which translates into reduced project risk. Furthermore, development time is significantly lower and less variable than either C++ or Java. Memory consumption is comparable to Java. Lisp thus presents a viable alternative to Java for dynamic applications where performance is important.

The experiment

Our dataset consists of sixteen programs written by fourteen programmers. (Two programmers submitted more than one program. This was also the case in the original study.) Twelve of these entries were written in Common Lisp [2], and the other four were in Scheme [3]. All of the subjects were self-selected volunteers recruited from an Internet newsgroup.

To the extent possible we duplicated the circumstances of the original study. We used the same problem statement (slightly edited but essentially unchanged), the same program input files, and the same kind of machine for the benchmark tests: a SPARC Ultra 1. The only difference was that the original machine had 192 MB of RAM while ours only had 64 MB, but since none of the programs used all the available RAM that should not have changed the results. Common Lisp benchmarks were run using Allegro CL 4.3. Scheme benchmarks were run using MzScheme [4]. All the programs were compiled to native code.

Results

Our results are shown in the accompanying figures, along with the data from the original Prechelt study. The results are presented as cumulative probability distribution functions. The Y-value at a particular point on the curve represents the fraction of programs whose performance on a particular metric was equal to or better than the X-value at that point. The horizontal

extent of the curve is an indication of the range of values. A smooth curve indicated evenly distributed values. A curve with discontinuous jumps indicates clustering of the data at the jumps.

Two striking results are immediately obvious from the figures. First, development time for the Lisp programs was significantly lower than the development time for the C, C+ and Java programs. It was also significantly less variable. Development time for Lisp ranged from a low of 2 hours to a high of 8.5, compared to a range of 3-25 hours for C/C++ and 4-63 hours for Java. The difference cannot be accounted for by programmer experience. The experience levels was lower for the Lisp programmers than for both the other groups (an average of 6.2 years for Lisp versus 9.6 for C/C++ and 7.7 for Java). The Lisp programs were also significantly shorter than the C, C++ and Java programs. The Lisp programs ranged from 51 to 182 lines of code. The mean was 119, the median was 134, and the standard deviation was 10. The C, C++ and Java programs ranged from 107 to 614 lines, with median 244 and mean 277.

Second, while execution times of the fastest C/C++ programs was faster than the fastest Lisp programs, the runtime performance of the Lisp programs in the aggregate was substantially better than C/C++ (and vastly better than Java[1]). The median runtime for Lisp was 30 seconds versus 54 for C/C++. The mean runtime was 41 seconds versus 165 for C/C++. What is even more striking is the low variability in the results. The standard deviation of the Lisp runtimes was 11 seconds versus 77 for C/C++. Furthermore, much of the variation in the Lisp data was due to a single outlier at 212 seconds (which was produced by the programmer with the least Lisp experience: under a year). If this outlier is ignored then the mean is 29.8 seconds, essentially identical to the median, and the standard deviation is only 2.6 seconds.

Memory consumption for Lisp was significantly higher than for C/C++ and roughly comparable to Java. However, this result is somewhat misleading for two reasons. First, Lisp and Java both do internal memory management using garbage collection, so it is often the case that the Lisp and Java runtimes will allocate memory from the operating system this is not actually being used by the application program. Second, the memory consumption for Lisp programs includes memory used by the Lisp development environment, compiler, and runtime libraries. This overhead can be substantially reduced by removing from the Lisp image those features that are not used by the application, an optimization we did not perform.

Analysis and Speculation

Our study contains one major methodological flaw: all the subjects were self-selected (a necessary expediency given that we did not have ready access to a supply of graduate students who knew Lisp). About the only firm conclusion we can draw is that it would be worthwhile to conduct a follow-up study with better controls. If our results can be replicated it would indicate that Lisp

---

[1] Newer versions of the JDK appear to improve Java runtime performance on this task by about a factor of 3 (Lutz Prechelt, personal communication). But because Java's original performance was orders of magnitude worse than the other languages this does not qualitatively affect our conclusions.

offers major advantages for software development: reduced development time and reduced variability in performance resulting in reduced project risk.

Our results beg two questions: 1) Why does Lisp seem to do as well as it does and 2) If these results are real why isn't Lisp used more than it is? The following answers should be considered no more than informed speculation.

When discussing Lisp's performance we need to separate four aspects which have potentially different explanations. First, there is the fact that Lisp's runtime performance appears comparable to C/C++. This result is contrary to the conventional wisdom that Lisp is slow. The simple explanation is probably the correct one: the conventional wisdom is just wrong. There was a time when Lisp was slow due to the unavailability or immaturity of compilers. Those days are long gone. Modern Lisp compilers are mature, stable, and of exceptionally high quality.

The second performance result is the low development time. This can be accounted for by the fact that Lisp has a much faster debug cycle than C, C++ or Java. The compilation model for most languages is based on the idea of a *compilation unit*, usually a file. Making a change to any part of a compilation unit requires, at least, recompiling the entire unit and relinking the entire program. It typically also requires stopping the program and starting it up again, resulting in a loss of any state computed by the previous version. The result is a debug cycle measured in minutes, often tens of minutes.

Lisp compilers, by contrast, are designed from the ground up to be incremental and interactive. Individual functions can be individually compiled and linked into running programs. It is not necessary to stop a program and restart it to make a change, so state from previous runs can be preserved and used in the next run rather than being recomputed. It is not unusual to go through several change-compile-execute cycles in one minute when programming in Lisp.

The third result is the smaller size of the Lisp code. This can be accounted for by two factors. First, Lisp programs do not require type declarations, which tend to consume many lines of code in other languages. Second, Lisp has powerful abstraction facilities like first-class functions that allow complex algorithms to be written in a very few lines of code. A classic example is the following code[2] for transposing a matrix represented as a list of lists:

```
(defun transpose (m) (apply 'mapcar 'list m))
```

The final performance result is the low variability of runtimes and development times. There are several possible explanations. It might be the result of subject self-selection. It might be because the benchmark task involved search and managing a complex linked data structure, two jobs for which Lisp happens to be specifically designed and particularly well suited. Or it might be because Lisp programmers tend to be good programmers. This in turn might be because good programmers tend to gravitate towards Lisp, or

---

[2] This example is considered a "parlor trick" by some because it is not something one would normally do in a real program due to its inefficiency. But it does serve to illustrate how concise the language can be.

it might be because programming in Lisp tends to make one a good programmer.

This last possibility is not as outlandish as it might at first appear. There are many features of Lisp that make it a very easy language to learn and use. It has very simple and uniform syntax and semantics. There is ubiquitous editor support to help handle what little syntax there is. The basic mechanics of the language can be mastered in a day. This leaves the programmer free to concentrate on the business of designing and implementing algorithms instead of worrying about the vagaries of abstract virtual destructors and where to put the semicolons. If Lisp programmers are better programmers it may be because the language gives them more time to become better programmers.

Which brings us to the question of why, if Lisp is so great, is it not more widely used? This has been a great puzzle in the Lisp community for years. If we knew the answer the question would be moot. One contributing factor is that when AI fell out of favor in the 1980's for failing to deliver on its lofty promises, Lisp was tarred with the same brush. Another factor is the dogged persistence of the myths that Lisp is big and slow. Hopefully this work will begin to correct that problem.

Conclusions

Lisp is often considered an esoteric AI language. Our results suggest that it might be worthwhile to revisit this view. Lisp provides nearly all of the advantages that make Java attractive, including automatic memory management, dynamic object-oriented programming, and portability. Our results suggest that Lisp is superior to Java and comparable to C++ in terms of runtime, and superior to both in terms of programming effort, and variability of results. This last item is particularly significant as it translates directly into reduced risk for software development.

References

[1] Lutz Prechelt. "Java vs. C++: Efficiency Issues to Interpersonal Issues." Communications of the ACM, October 1999.

[2] Guy L. Steele. *Common Lisp, the Language, second edition*. Digital Press, 1990.

[3] William Clinger and Jonathan Rees, editors. "The revised^4 report on the algorithmic language Scheme." *ACM Lisp Pointers* 4(3), pages 1--55, 1991.

[4] Matthew Flatt. "MzScheme Language Manual." http://www.cs.rice.edu/CS/PLT/packages/mzscheme/

Acknowledgements

Thanks to Lutz Prechelt for making available the raw data from the original study, and to Dan Dvorak for calling the Prechelt study to my attention. Lutz Prechelt, Dan Dvorak and Kirk Reinholtz provided comments on an early draft of this paper. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration.

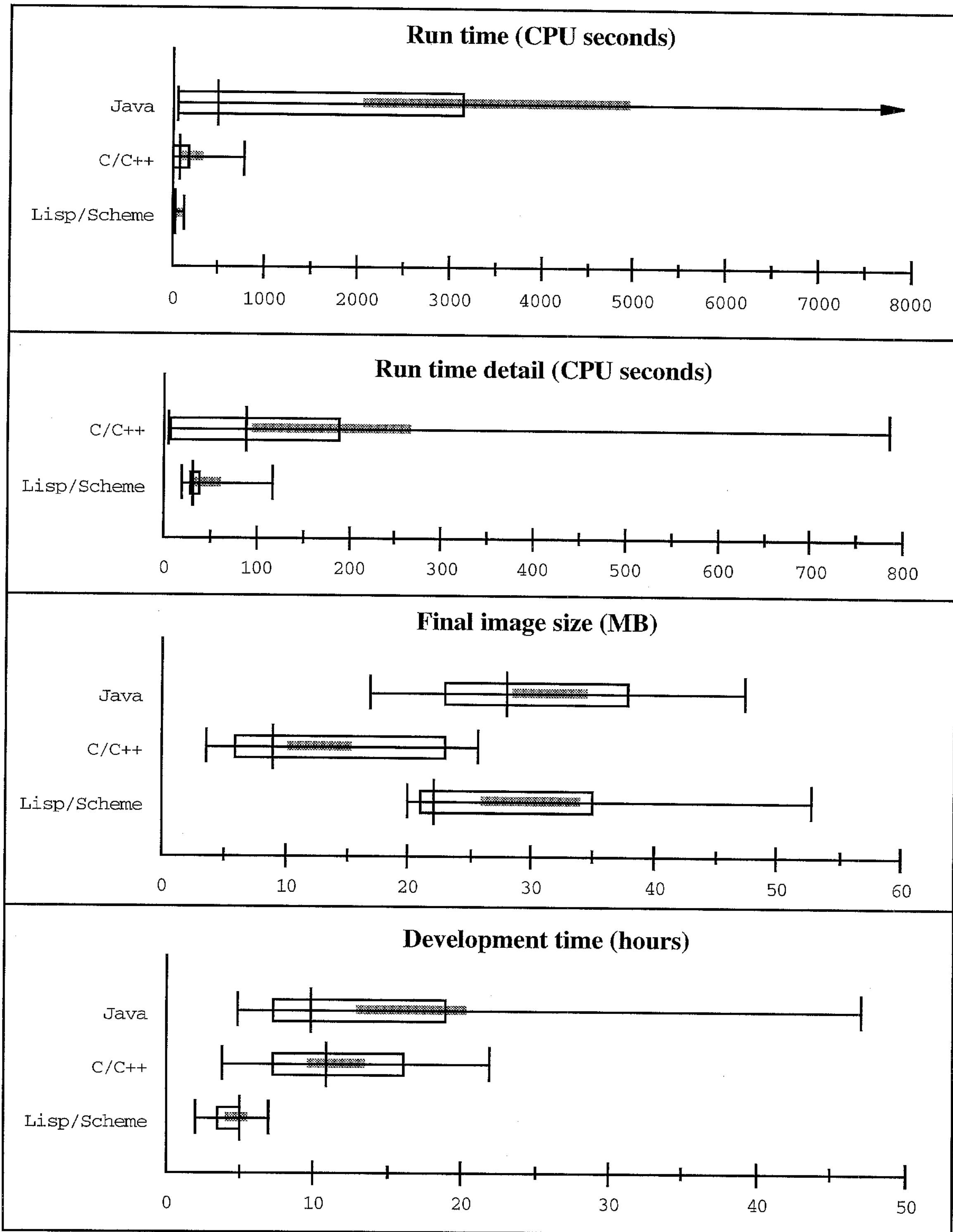

Figure 1: Experimental results. The vertical lines from left to right indicate the 10th percentile, median, and 90th percentile respectively. The hollow box encloses the 25th to 50th percentile. The thick grey line is the width of two standard deviations centered on the mean.