AN EMPIRICAL EVALUATION OF THE JAVA AND C++ PROGRAMMING LANGUAGES

David R. Pratt, Anthony J. Courtemanche, Jamie Moyers and Charles Campbell
Science Applications International Corporation
Orlando, Florida

Abstract

The scarcity of applicable empirical data on the issue of C++ versus Java performance led the authors to conduct their own series of performance studies. A performance comparison was made of Java and C++ in the implementation of a test system representative of those encountered in simulation systems. The algorithms chosen were deemed to be representative of both the algorithms used in simulation systems and those which consume the majority of the time-based computational load. They included a hull dynamics model, geometric intervisibility, and a scheduler / dispatcher. The exact same algorithms were implemented in both Java and C++.

As with some developmental programs, execution speed was not the only item of concern in this study. Often programmer productivity and error rates are major factors in choosing a particular programming language. To capture information about these factors, productivity rates of each of the programmers were recorded as they developed code from scratch and ported the code to the new language. Subjective evaluations from each of the programmers concerning their opinions on the ease of using the language for the given applications was also collected. This paper describes the programming language study and presents empirical and subjective findings that both program managers and developers should be aware of when making programming language selections for future simulation systems.

This is the second paper in a series of empirical studies the authors have conducted into the relative performance programming languages and their suitability to the Modeling and Simulation Domain.

Author Biographies

Dr. David R. Pratt is a Chief Scientist/Fellow at SAIC's Applied Software Systems Engineering Technology (ASSET) Group. As the Group's Technical Lead, he coordinates the internal research activities of the group and helps to set the technical directions. Prior to joining SAIC, he served as the JSIMS Technical Director, a tenured Associate Professor of Computer Science at the Naval Postgraduate School, and a Captain in the Marine Corps. Dr. Pratt received a Ph.D. in Computer Science in 1993 and a Masters of Science degree in Computer Science in 1988 from Naval Postgraduate School.

Anthony J. Courtemanche has over ten years of experience in virtual entity simulations utilizing Computer Generated Forces, with contributions in software architecture, weapons systems simulation, targeting behaviors, network simulation protocols, and user interfaces. Mr. Courtemanche was one of the principal contributors to the ModSAF architecture and was the Project Engineer for ModSAF system development. At SAIC, Mr. Courtemanche is a Chief Scientist in the Advanced Distributed Simulation Research Team and supports research projects in the areas of object-oriented software architectures, advanced software technologies, and computer generated forces. Mr. Courtemanche received his Masters of Science in Electrical Engineering and Computer Science from MIT in 1987.

Charles Campbell is a senior software engineer assigned to SAIC's ASSET Research Programs division. His responsibilities include design and development of Synthetic Natural Environment representations and services for CGF systems. Mr. Campbell holds a Masters of Science in Computer Science from the University of Central Florida and a Bachelor of Science in Computer Science from Indiana University. His interests include simulation, software engineering, computational geometry, and computer graphics.

James Moyers is a software engineer assigned to SAIC's ASSET Research Programs division. Mr. Moyers holds a Masters of Science in Computer Science from the University of Missouri - Rolla (UMR) and a Bachelor of Science in Computer Science from Southeast Missouri State University. His interests include computer graphics, visualization, object oriented programming, and image processing.

AN EMPIRICAL EVALUATION OF THE JAVA AND C++ PROGRAMMING LANGUAGES

David R. Pratt, Anthony J. Courtemanche, Charles Campbell, and James Moyers
Science Applications International Corporation
Orlando, Florida

INTRODUCTION

When C++ and Ada first came into widespread use, many simulation developers felt they would not be suitable as development languages for simulations due to their perceived slower execution speed when compared with C and FORTRAN. However, both languages have been used quite successfully on large-scale systems. A similar situation now exists with Java, the new programming language rave. Its use is widespread in the commercial sector and the market forces behind Java are developing feature and performance enhancements rapidly. Nevertheless, many traditionalists are concerned about using Java in simulations due to perceived performance issues.

Despite the current popularity of both C++ and Java as development languages and frequent discussions comparing the two, the authors were surprised to find little applicable empirical data on the issue of C++ versus Java performance. This scarcity of data led the authors to conduct a series of studies comparing the performance of Java and C++. The studies focused on implementing test systems that are representative of those encountered in simulations. While the first study focused on the comparison of the programming languages across a series of computational platforms (Pratt, 2000), this study examines the different optimization and execution modes of the sample programs. This paper describes the programming language study and presents empirical and subjective findings that both program managers and developers should be aware of when making programming language selections for future simulations.

OBJECTIVES

The goal of this study was to gather scientifically defensible data to allow a comparison of Java and C++ with respect to development and runtime performance in simulations. Four basic objectives were identified:

- Objectively evaluate the performance of Java and C++ as applied to the development and execution of simulations
- Evaluate the optimization options and their effect on run times
- Gather subjective evaluations of the Java and C++ languages and development issues
- Logically extrapolate the results for future simulations

As with some developmental programs, execution speed was not the only item of concern in this study. Often programmer productivity and error rates are major factors for choosing a particular programming language. To capture information about these factors, productivity rates of each of the programmers were recorded as they developed code from scratch and ported the code to the new language. Subjective evaluations from each of the programmers concerning their opinions on the ease of using the language for the given applications was collected in addition to objective performance data.

STUDY SETUP

To achieve the above objectives, we selected a series of different compiler and runtime optimizations and representative simulation components. The next few sections describe the study setup.

Computer Platform

To minimize variable results caused by the use of different computer platforms, a single computer platform was used in this study. The designated system was a Dell Latitude CPi running Windows 95 with a 300 MHz Pentium II CPU and 128 MB of RAM. The computer was disconnected from the network and the screen saver was turned off to minimize computational variations during the runs.

Development Environment

This effort focused on evaluating the two programming languages independently of any Integrated Development Environments (IDEs) that might be used to generate, compile and/or run the code. We decided to use a combination of in-house, free, and evaluation software as the development environment. This actually duplicated the normal development environments of most of the members of the evaluation team (i.e., the developers).

Java Due to Sun's effective lock on the Java Standard, the Java Development Kit (JDK) is the reference implementation for Java. Fortunately, it is available free of charge from www.javasoft.com. The version used for this

project was JDK 1.3. The "`java -version`" command returned the following result: *`Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0-C) Java HotSpot(TM) Client VM (build 1.3.0-C, mixed mode).`*

One of the key features of this JDK version is the Just In Time (JIT) compiler. Besides doing some code optimizations when a class is loaded at runtime, the JIT compiler speeds performance by compiling Java byte-code into a native instruction set for the machine. The result is stored in an instruction cache similar to how natively compiled code would be stored. A more complete discussion of the Java JIT can be found in (McManis). More details on the HotSpot™ VM can be found in (Sun).

C++ Maintaining the freeware philosophy and our desire for cross-platform compatibility, we chose to use the GNU C++ compiler for this study (GCC). The GNU C++ compiler is part of the GNU Compiler Collection (GCC). The particular version used was *`gcc version egcs-2.91.57 19980901 (egcs-1.1 release)`*. GCC is available from gcc.gnu.org.

Constraints

To avoid the problems that plagued Precheltr's study (Precheltr, 1999) (namely, different programming styles), we decided to levy some constraints on the developers participating in this study. A detailed discussion of the programming constraints is contained in 0. To summarize, the imposed constraints were:

- Consistent implementation across platforms and programming languages
- Solid and explicit coding style
- No visual/graphical components

While some of the developers felt that the constraints skewed the study in favor of Java, which was considered to be the simpler of the two languages, the majority of the developers viewed the constraints as being the only way to ensure a fair comparison between the programming languages. A discussion of the differences between the languages can be found in (Thimbleby 1999).

STUDY APPROACH

Three of the four performance tests from the previous study were used again in this study:

- Line of Sight (LOS) Computation
- Dynamics Modeling
- Scheduler

Once the initial reference implementations were completed, the code was given to the other two developers to be rewritten in the other language. The language implementations were designed so that a line for line translation could be done to ensure the consistency of the results and execution semantics. After the coding and debugging phases were completed, a peer review was conducted to verify that the implementations were consistent and that they produced the same results.

In the production phase of the project, each of the three tests was run ten times using the Java Virtual Machine (JVM) / C++ compiler command arguments shown in Table 1. The results were averaged over the number of runs. If a value was more then 30% away from the mean, it was discarded along with the opposite minimum or maximum. This happened in only two cases. The final results were then analyzed for trends and insights.

Java	C++
`java`	`g++ -Wall`
`java -Xint`	`g++ -Wall -01`
`java -Server`	`g++ -Wall -02`
	`g++ -Wall -03`

Table 1. Command Options Used

Java Test Cases

The same Java class files were used for all three Java cases. The only difference among the cases was the execution mode of the programs. In the first case, the system was run in a mixed mode. In mixed mode, the most heavily executed code was compiled to machine code at runtime by the JIT compiler, which was part of the JVM. The mixed mode is the default mode for the JDK 1.3.

The second "`-Xint`" mode was a non-standard JVM option that did not involve any JIT compilation of code. Instead, the JVM operated strictly in the interpreted mode on the source byte code. As a result, no code optimizations were performed as part of the execution. In both the first and second cases, the JVM ran in the client mode. The client mode is optimized for smaller programs and allows quicker program startup.

The final Java mode was the HotSpot™ server mode invoked using the "`-Server`" option. In this mode, the program's byte code was compiled more aggressively and optimized for long periods of execution. Note, it is still a mixed mode execution. Use of the HotSpot™ server mode entailed additional compilation, optimization of the memory allocation process, and better threading performance. The tradeoffs for using this mode, however, in-

cluded a longer startup period and a larger memory footprint.

C++ Test Cases

For all the C++ cases, the "– Wall" option forced notification of all warnings generated by the compiler. The option did not affect the runtime performance of the system though. The different C++ modes stemmed from using the GNU C++ compiler with four different levels of performance optimization as discussed below.

The first C++ mode involved no optimizations. High level C++ instructions were converted directly to machine code with no attempt to allocate variables to registers or to merge any of the instructions. Since there was no real buffering or modification to the code during compilation, this was the fastest and least resource intensive compilation mode. This mode also provided the best debugging support because there was a well established, independent mapping between the machine code and the C++ code statements.

The second "–O1" compiler mode involved optimizations that could be done quickly. Principally, variable allocations to registers and attempts to reduce code size and execution time are done. The compiler also attempts to optimize the branching and function calling processes.

The third "–O2" compiler mode involved additional optimizations beyond those of the "–O1" compiler mode. Typically the compiler invokes nearly all the supported optimizations that do not involve a space-speed tradeoff such as inlining functions or unrolling loops. The compile process generally takes longer, but it should generate faster executing code. The optimizations do increase the difficulty of the runtime tracing and debugging process, however.

The fourth and last "–O3" compiler mode involved all possible optimizations to reduce the execution time of the program. Inlining functions and unrolling loops may be done. Resulting program executables tend to be larger in size, but they have shorter execution times (space / speed tradeoff).

It is interesting to note that since the GNU C++ compiler supports a wide range of platforms, the optimizations that are performed on one platform may or may not be performed on another platform. Thus, the optimization characteristics may vary by machine and operating system.

TEST SUITE

The three different performance tests comprised a test suite that allowed a wide range of data to be collected. Each of the tests is described further in this section along with it associated performance metric.

LOS Computation Test

One of the most common computations carried out by a simulation is the Line Of Sight (LOS)[1] calculation. Hence, it was one of the key algorithms included in our performance test suite. Graphically depicted in Figure 1, we chose a simple LOS algorithm that is based on a regularly gridded elevation matrix. It returned a boolean value to indicate whether intervisibility existed between a given start point (viewer) and end point (target).

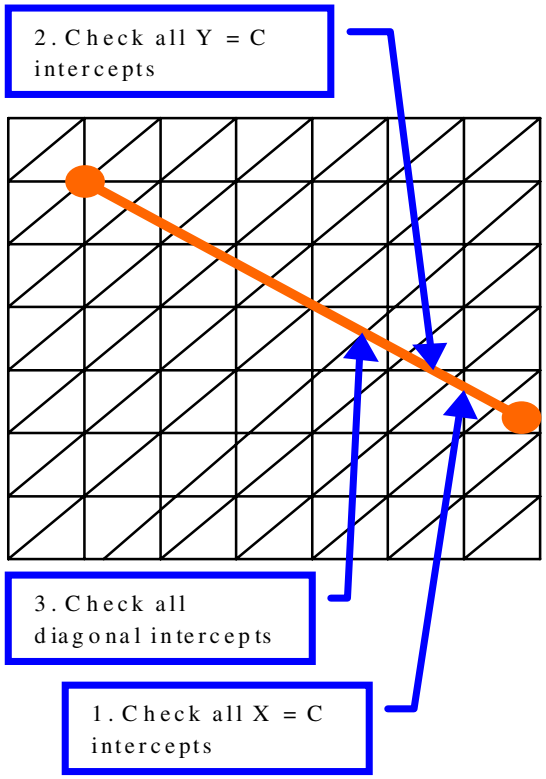

Figure 1. Line of Sight Algorithm

The algorithm consisted of three loops. The first loop stepped through all of the north-south (X = Constant Value) divisions to determine if the point on the line segment connecting the viewer and target end points was below the terrain skin. If so, the LOS method ceased all further processing and returned a value of **FALSE**. After stepping though all of the north-south divisions, a similar loop was used to step through and check all of the east-

[1] Some purists prefer the term geometric intervisibility (GI) for a calculation that deals only with binary point to point occlusion by the geometric components of the terrain data base. We are using the more common term of Line of Sight (LOS) without any loss of generality.

west (Y = Constant Value) divisions. Then lastly, all of the diagonals were considered and checked. If all three loops executed to completion, the LOS method returned a value of **TRUE**, which implied intervisibility does exist between the two given points.

The terrain database used for the study was constructed from a Digital Elevation Terrain Data (DTED) level 0 sample of 121 by 121 posts. The intervals between postings were set to a constant 100 units value. A uniform right triangulation was imposed with the diagonals going from the lower left to upper right corners of each grid box.

A driver class was used to run the test. A sample set of 200,000 segments, each 60,000 units long, was used for the LOS test. The start point and the direction of the segment along which LOS was to be computed were randomly generated. If the computed end point was outside the bounds of the terrain sample, the direction was adjusted until the end point was within the bounds.

The LOS test driver set the initial eye points and targets to zero units above the terrain. The sample set of 200,000 segments was then tested. If over half of the segment tests returned **TRUE** (e.g., LOS existed between the eye point and the target), then the program exited. If less then half returned **TRUE**, then the eye points and targets were raised 100 units and the 200,000 segments were tested again. For the sample database, over half of the segments tested returned **TRUE** when the eye points and targets were 700 units above the terrain. As a result, 1,600,000 LOS calculations were done for each timed run. The collected performance metric was elapsed time for all of the LOS test runs.

Dynamics Modeling Test

One of the major applications of polymorphic method invocation in continuous and time stepped simulations is the use of the *vehicle.tick(deltaTime)* construct to abstract the details of the entity movement routines. For this reason, we designed the Dynamics Modeling test to implement three of the most common ways of modeling an entity's motion. The test involves the use of a single base class, the HullClass, and three derived child classes: StaticClass, KinematicClass and DynamicClass (see Figure 2).

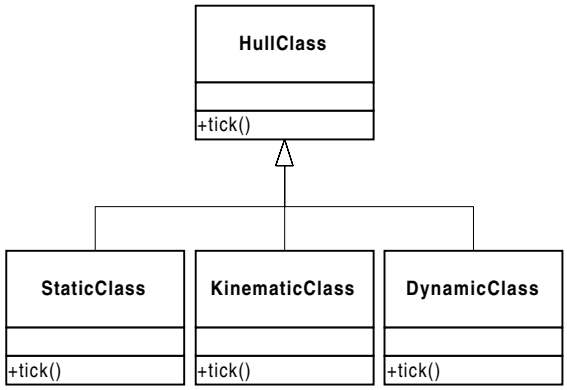

Figure 2. The HullClass Hierarchy

The StaticClass performed no actual movement computations and was used primarily as a means to compute the overhead costs associated with using the method. Movement for the KinematicClass was determined algorithmically based on commanded and actual values for orientation and velocity. Movement for the DynamicClass was based on a summation of forces using Newtonian physics. In all cases, interactions with the terrain were not taken into account. The performance metric collected for the Dynamics Modeling test was elapsed time using the three HullClass child classes.

A second set of runs were made for the Dynamics Modeling test using a modified version of the Java code. During these runs, "scratch" vectors were allocated as part of the classes to eliminate the requirement for the system to allocate space to handle the intermediate values used in the vector computations. The algorithms, control structures, and geographic ending locations of the hulls did not change from those of the initial code version runs, however.

For the Dynamics Modeling test runs, a driver program was created to simulate two days of simulation time. With a tick rate of 15 Hz, this amounted to 2,592,000 calls to each of the vehicle movement routines. For each run, the driver program exercised each of the three different leaf classes separately, reported the elapsed time for each, and then summed the results for the run.

Scheduler Test

The Scheduler test was an implementation of a Discrete Event Simulation (DES) As Fast As Possible (AFAP) control structure. It used a singly linked list as the event queue. As an initial condition, it was seeded with 200 events, of which 25% were Recurring Events and 75% were Single Events. When a Recurring Event fired, it was rescheduled at a constant interval based on an initial random draw in the constructor between 1 and 100 time units. When a Single Event fired there was a 25% chance

that no new event would be scheduled, a 50% chance that one event would be scheduled, and a 25% chance that two independent events would be scheduled.

All Single Events were scheduled based upon a random draw between 1 and 100 time units in the future. The system was set to run for 50,000 time units and resulted in the firing of 145,402 events. Again, elapsed time was the performance metric used for all the Scheduler test runs.

RESULTS

This section presents a subset of the results from this program language study. Though informative, source data spreadsheets and graphs of all three performance test results were too voluminous to be included in this paper. The subset presented below is representative of the study and is not biased toward any of the conclusions.

LOS Computation Test Results

In the LOS test runs, the relative performances of the two languages were about what we had expected based upon our previous work comparing them. The results from the various runs using the different execution and optimization modes were new and interesting, however. Figure 3 shows that by simply activating the first level of compiler optimizations (OPT1), the C++ program ran 27% faster than it did with no optimizations (noOPT). While we had expected an increase in performance, we did not expect it to be quite so significant. Surprisingly, the other two C++ optimizations modes did not produce as large an increase in performance. Use of both the "–O2" and "–O3" optimization modes resulted in only a 26% improvement in the run times over the non-optimized case. We believe this can be attributed to the relative simplicity of the calling and branching structure of the code and the dynamic nature of the `For` loops.

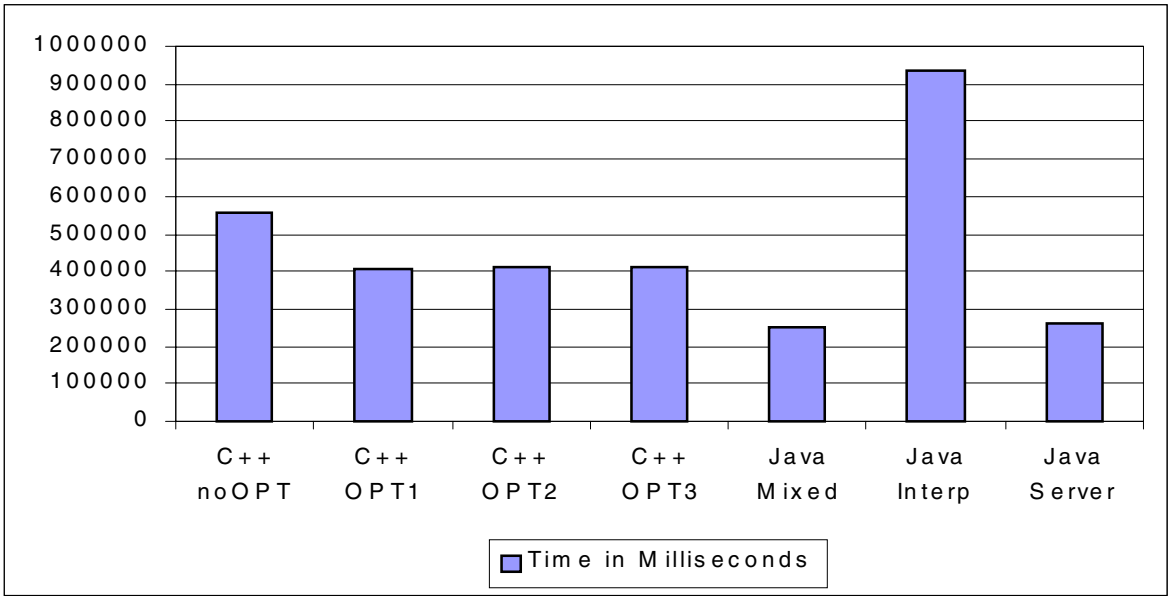

Figure 3. Line of Sight Computation Test Results

The mixed mode Java implementations (labeled "Java Mixed" and "Java Server" in Figure 3) were observed to perform better than C++ implementation. This can be attributed to the type of coding and program structure used for this study. Namely, the type of coding was better suited for Java JIT optimizations than it was for the C++ compiler's optimizations. However, the biggest factor was most likely use of the JIT compiler. The large number of repetitive calls to compute LOS and the fact that the LOS algorithm had a high degree of locality of reference (i.e., code located in a small loop) contributed to increased JIT performance. A detailed discussion of the HotSpot™ JIT performance engine architecture and the optimization process is presented in (Sun, 1999).

The effect of the JIT can be seen most clearly by comparing the results of the two Java mixed mode runs to the results of the Java interpreted mode runs. On average, the mixed mode run times were approximately 27% to 28% of the interpreted mode run times

Dynamics Modeling Test Results

Table 2 shows the results for the Dynamics Modeling test runs. It is interesting to see the meager performance improvements exhibited by the three C++ cases involving optimizations, only 1% to 4%. These results are in stark contrast to the performance increases seen before for the LOS test. This finding emphasizes the point that results

of performance metrics analyses are dependent on how the performance metrics are generated. There are limits to the inferences that can be drawn regarding the domain of applicability for such results. Too often, the limits are not fully understood and misleading, blanket statements are made.

Test	Static	Kinematics	Dynamics	Total	Normalized
C++ noOPT	429	5614	31899	37942	100%
C++ OPT1	258	5986	31364	37608	99%
C++ OPT2	257	5948	31226	37431	99%
C++ OPT3	247	5620	30496	36363	96%
Java Mixed	550	10995	38344	49889	131%
Java Interp	3389	64642	272518	340549	898%
Java Server	5644	8394	31001	45039	119%
Java Mixed 2	167	2800	18620	21587	57%
Java Interp 2	805	16072	92841	109718	289%
Java Server 2	1652	4730	16665	23047	61%

Table 2. Dynamics Modeling Test Run Times in Milliseconds

Consistent with the results obtained in (Pratt, 2000), the first set of Java runs performed slower than the C++ runs. In our first study, we attributed this to the memory allocations performed as part of the computations. We felt that this was justified based upon the very high cost associated with doing memory allocations in Java. This trend was quantified in the Object Oriented test runs conducted in our previous work.

The results from the second set of Dynamics Modeling test runs using Java classes which included scratch vectors are shown in tabular form in Table 2 and in graphical form in Figure 4. There was a significant improvement in the runtime performance just by having the pre-allocated scratch vectors. In Figure 4, the bar representing the Java interpreted mode run time without the scratch vector (labeled “Java Interp”) actually extends beyond the limits of the graph’s vertical axis by a factor of more than three.

When comparing the strictly interpreted Java runs to each other, the pre-allocated vectors cut the run times of the second series of runs by 68%, one of the biggest performance differences of any of the comparisons conducted on this study. In terms of machine performance, the increase is roughly equivalent to the use of a 900 MHz machine over a 300 MHz machine. In the Java mixed mode and server mode cases, the run times were 43% and 51%, respectively, of the corresponding original run times. These improved times were still quite significant, but not as dramatic as the increase seen with the interpreted mode runs.

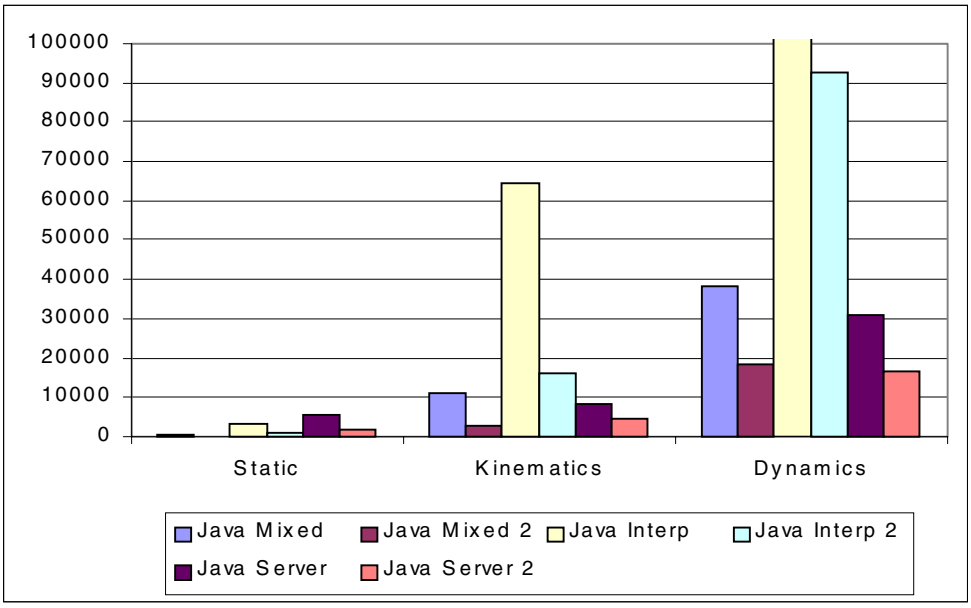

Figure 4. Comparison of the Two Java Implementations of the Dynamics Modeling Test

Scheduler Test Results

Of all the test runs conducted in this study, the Scheduler test runs had the shortest run times. The results of these runs highlighted a drastic difference in the startup costs between the mixed mode and server mode implementations of the JVM. Specifically, there was simply not enough run time to recoup the up front time investment made in doing the optimization. Figure 5 shows that using the server mode JVM was roughly 11 times more expensive than using the mixed mode JVM. Using the server mode JVM was also 1.75 times more expensive than using the plain Java interpreter.

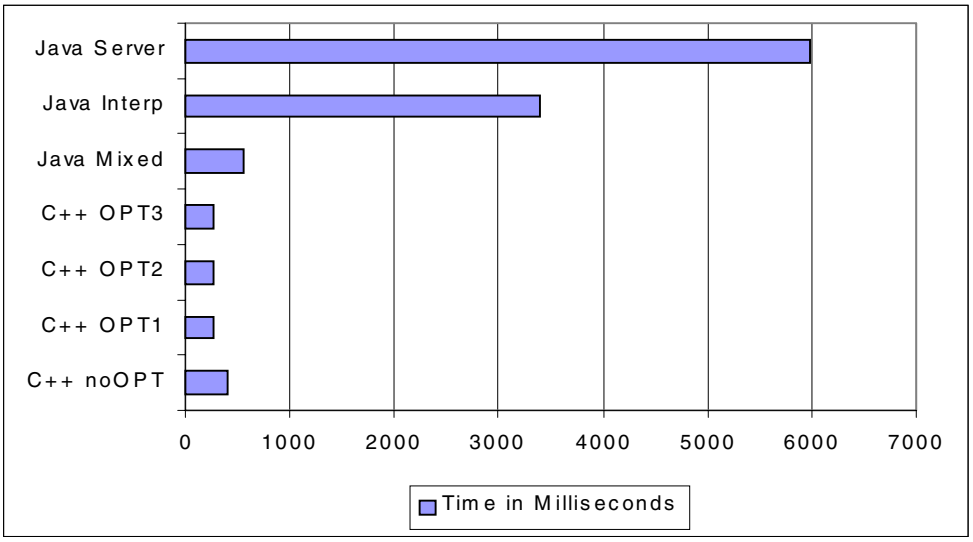

Figure 5. Scheduler Test Results

Interestingly, the C++ implementations provided better performance than the Java implementations in all of the Scheduler test cases. This counters the results obtained in a similar test in our previous study. Because the test in the previous study had a smaller number of events and was shorter in duration, the scheduler's queue was significantly smaller. From this we might be tempted to hypothesize about the languages' relative abilities to handle large data structures. However, further study is needed before any definitive conclusion can be drawn on this issue.

Subjective Language Evaluations

The two languages were also subjectively evaluated by the study team. All of the developers had significantly more experience with C/C++ than Java. However, with one exception, the developers preferred coding in Java rather than C++. For example, one of the developers commented that he hated to have do all the "tricks" necessary to write code in C++. The single dissenting developer, who was perhaps the best C++ programmer but also the least familiar with Java, had internalized the C++ "tricks" and felt constrained that he could not do similar things in Java. Nearly all of the developers perceived Java to be a safer and cleaner programming language.

There was not a significant difference in the development time between the two languages for these simple programs. However, there is documented anecdotal evidence that Java is both a safer and quicker development environment (Phipps, 1999). In fact, during part of this study, we noticed that the C++ version of the Scheduler tests had astronomical run times. Upon further investigation, we found a memory leak that had gone undetected.

CONCLUSIONS

In this paper, we have presented a scientifically defensible argument for the use of Java relative to C++ based on performance considerations. We caveat our findings by reiterating the scope of programming language study. Namely, the study was designed to compare the performance and use of Java and C++ as programming language options for simulations. It involved a relatively simple set of test programs and test cases targeting components of simulations that are commonly considered to be computationally intensive. The simulation specific tests were run across a range of optimization options and execution modes. From our validity assessment and analysis of the data, especially when it is combined with the information in (Pratt, 2000), (Price) and (Rijk), we feel confident that the results can be generalized and used as a point of departure on which to base system level decisions. The study results, however, like all benchmarks, are only relevant to the domains from which they were taken.

Our findings are consistent with the results from (Precheltr, 1999), which found that the differences in programmer ability were more significant then any difference in programming language. We have shown in this study that the developer's knowledge of the compilers and run time environments can have a much greater bearing than the choice of language. In one case, a simple change in command line options and memory management resulted in a 15-fold increase in performance. The poor performance of the Java interpreted modes, along with the fact that the initial JVMs ran strictly in interpreted modes, are what gave Java its reputation as a slow language. But the results of our programming language studies show that use of the JIT puts Java in the same performance league with C++.

When making programming language decisions for future simulations, one cannot overlook the large investment in legacy simulations written C/C++. C++ is a relatively stable and mature programming language. Java, on the other hand, is a relatively new language, and it is still undergoing many performance and feature improvements. As mentioned in our previous study, we believe that Java will continue to improve to a point where performance is no longer an issue. Market forces behind Java continue to develop significantly faster and more efficient JIT compilers and JVMs and other considerations such as Java programming ease, portability of legacy source code to Java, and the advantages of an interpreted language further Java's continued growth and popularity. The results of this study have affirmed our expectation that Java will overtake C++ as the language of choice for many applications, including simulations, in the near future.

ACKNOWLEDGMENTS

This work was done under SAIC internal research funding.

REFERENCES

GCC Home Page, http://gcc.gnu.org/

McManis, Chuck, "Just In Time Compilation", http://www.javacats.com/us/articles/chuckmcmanis091696.html

Phipps, Geoffrey, "Comparing Observed Bug and Productivity Rates for Java and C++," Software - Practice And Experience , 29(4), 345–358 (1999)

Pratt, David, et al. "An Emperical Evaluation of Programming Languguages for Computer Generated Forces," 9th Conference on Computer Generated Forces and Behavioral Representation, May 2000 Pages 151-161.

Precheltr, Lutz, "Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences", Communications of the ACM, Volume 42 , Issue 10 (1999), Pages 109-112.

Price, John W., "Programming Language Performance Testing," http://www.r2systems.com/LangTest/

Rijk, Chris, "Binaries Vs Byte-Codes," http://www.aceshardware.com/Spades/read.php?article_id=153

Sun Microsystems, Inc. “Java HotSpotTM Server VM 2.0” Web Site, http://www.javasoft.com/products/hotspot/

Sun Microsystems, Inc. "The Java Hotspot Performance Engine Architecture," http://java.sun.com/products/hotspot/whitepaper.html, April 1999

Thimbleby, Harold, "A Critique of Java," Software - Practice And Experience , 29(5), 457–478 (1999)