Java and C Performance Comparison on Palm OS

Zhi-Kai Xin
zxin@cs.columbia.edu

Abstract

This paper investigates the performance comparisons of Java and C on Palm OS PDA device. The performance comparison concentrates on the memory management and the numerical computation of both languages. Execution time and memory usage are used as the measurements. The modern PDA and wireless devices offer powerful applications, so the choice of using Java or C becomes very important. This paper also addresses the doubt on whether Java is an acceptable embedded system language.

1 Introduction

Java has gained popularity over last couple years. It is no longer a web language. Many back-end servers have deployed with Java technologies. With the introduction of Java 2 Enterprise Edition (J2EE) on existing Java 2 Standard Edition (J2SE), Java has become the only E-commerce solution. The standard Java library APIs offer fast and easy application development. The portability of Java code allows the evolvement and enhancement of software. Java's memory management provides automatic garbage collection that allows safe software development. Modern computing has put more focus towards Personal Digital Assistance and wireless phone devices. Typical PDA device contains calculator, memo pad and calendar applications. Other high-end PDA contains mpeg and mp3 players. These applications need both powerful hardware and software support. In the current market, PDA uses Palm OS, Pocket PC, and Window CE as the most popular embedded OS. This project chooses Palm OS power PDA because Palm OS offers more development tools and Opensource software. Both Java and C runs on Palm OS based PDA devices. There are many literature surveys on C/C++ and Java performance comparison. Although C offers better memory usage and execution time performance but Java's performance has been improved with newer releases and implementations of the JVM. Just-in-time (JIT) Java compiler for the latest J2SE can turn Java byte-code into native machine code during runtime, so it can significantly speed up Java performance. Many researches have also suggested that by rewriting some of the existing Java software in more efficient manner can greatly improve the Java performance. Sun Microsystems offers Kilo-byte Virtual Machine (KVM) for Palm OS powered PDA devices. KVM is a stripped down version of JVM. The project measures the performance of Java and C running on Palm OS. The performance is measured in terms of memory usage and execution speed. Does Java offer acceptable performance on Palm OS based PDA device? What improvements can be done from programmer side? These are the main questions the paper tries to address. The rest of the paper is organized as follows – Section 2 describes some existing related works. Section 3 describes the project plan. Section 4 describes the results. Section 5 is the conclusion and Section 6 is the future works.

2 Related Works

2.1 KVM for Palm OS

In June 1999, Sun Microsystems released Java 2 Micro Edition[1] (J2ME). It is targeted for PDA and wireless devices where power consumption and memory are very stringent. J2ME is divided into following layers [1][2][3]:

- Kilobyte Virtual Machine (KVM).
- Configurations. *Connected Device Configuration* (CDC), *Connected Limited Device Configuration* (CLDC).
- Profiles. *Mobile Information Device Profile* (MIDP).

KVM is a slim version of JVM that requires about 80 Kilobyte of memory. Java byte code such as .class or .jar files can be run on KVM. CLDC defines the standard Java platform for wide range of PDA and wireless devices. CLDC is also the specification of JVM that can be run on particular range of devices [2]. It is also responsible for delivery of Java applications to the devices. MIDP is more specific subset of CLDC targeting particular kind of PDA or wireless devices. MIDP is the Sun Microsystems' JVM implementation targeting Palm OS devices. ChaiVM of HP is a JVM targeting Pocket PC based PDA devices. KVM differs from JVM that it lacks of following features [1]:

- Floating Point Math**.** No float variable.
- Java Native Interface (JNI).
- Custom Class Loader.
- Reflection and Introspection.
- Thread Groups.
- Finalization.

Typical architectural hierarchy of J2ME looks like:

MIDP
CLDC
KVM
Host Operating System

[1] http://java.sun.com/j2me/

Figure 1. J2ME architecture

Developer should directly interact with MIDP library. The Host Operating System is Palm OS in this project.

2.2 Smart Object Management

Sosnoski [4] analyzed the performance of Java and C/C++ with various compilers and JVM implementations. The results showed that C outperforms Java in memory usage and execution speed. Java's automatic memory management handles all the memory allocation and de-allocation without developer's intervention but it also creates an extra overhead to the Java software. Due to this extra overhead, Java object's memory usage is rather very high [4]:

	Content (bytes)	JRE 1.2.2 (Classic)	JRE 1.2.2 (Hotspot 2.0 beta)
java.lang.object	0	28	18
java.lang.Integer	4	28	26
Int[0]	4	28	26
java.lang.String (4 characters)	8+4	60	58

Figure 2 memory usage in (bytes)

According to Sosnoski [4], different JVM implementations show very unique memory allocation usages. Newer version of JVM does give much better performance. Memory usage is only one problem with Java object. Its allocation time is also worth notice [4]:

Matrix size	Java	C/C++	FORTRAN
64x64	2.2	137.6	205.4
500x500	1.6	91.1	193.3

Figure 3 Memory Management Performances (time in seconds)

According to Figure 3, the memory allocation time of C/C++ is apparently much better than Java, although newer version of JVM does give more acceptable memory allocation time. In order to improve Java performance, Sosnoski [4] suggested modifying the Java code to use more primitive Java types instead of Java objects. Java contains primitive types of boolean, byte, char, double, float, int, long, and short. Developer should avoid using their wrapper classes such as Integer, Double, Long, Short, etc… Wrapper class represents immutable values of the corresponding primitive types, which give extra memory and performance overhead [4]. Utilities classes such as java.util.Vector and java.util.Hashtable should also be avoided as much as possible since each element must contain a Java non-primitive object or custom object. For instance, java.awt.Point class is used to represent a 'point' such as x and y coordination. Sosonoski [4] suggested using Java primitive type long to represent a Point. Since long is 64 bit in size so the higher bits can represent x coordinate while the lower bits can represent y coordinate[2]. Sosnoski [4] also suggested using dedicated object reuse and object pool concepts to avoid creating new object every time when the object is used very frequently. Database connection object or file descriptor object should only be created once and rest of the program should just reuse those objects without re-creating them again.

2.3 Benchmark test for Java and C/C++

Sosnoski [5] carried out series of benchmark testing on various Java compilers and JVM implementations such as HotSpot JVM and IBM win32 JRE. The benchmark test areas include:

- Basic numerical computation
- File I/O
- Memory management
- Typecasting overhead
- Multi-thread and Synchronization

The benchmark test results showed C/C++ definitely outperforms Java in many aspects. But newer version of Java compiler and JVM does improve the overall Java performance. IBM win32 JRE actually outperforms C/C++ in numerical computation by small percentage.

2.4 Numerical Computation

	JRE 1.2.2 (Classic)	JRE 1.2.2 (Hotspot 2.0 beta)	C/C++
331MB	26	14	9

Moreira et al. [6] compared the matrix multiplication benchmark test with Java, C/C++ and FORTRAN. Here are the results:

Figure 4. Performance measured in Mflops

FORTRAN and C/C++ clearly outperform Java in matrix computation. The matrix is implemented using array in all three languages. Java has an overhead of array checking where extra code is inserted to test array boundary and array index validity. Java throws indexOutOfBound

[2] For code example on representing Point with long, see http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html

exception if the software tries to access invalid array index or null array. Another problem is that Java does not have true multidimensional array, instead it has array of arrays. C/C++ and FORTRAN use true multidimensional array so the indexing is much faster. Moreira et al. [6] proposed to disable Java runtime array checking mechanism and Java's matrix multiplication performance got 15-fold improvement. Many Java runtime features must be left out in order to improve the overall performance.

Boisvert et al. [7] also pointed out the problems of multidimensional array with Java in numerical computation. Getting rid of Java runtime array checking was the solution proposed to improve the matrix multiplication performance. Complex number is also popular in numerical computation, Java implementation of complex number incurs overhead of object accessing. Boisvert et al. [7] presented a list of Do's and Don't for numerical computation in Java in order to improve its performance:

- Do use latest and modern JVM
- Do alias multidimensional array that is turn A[i][j][k] to Aij[k]
- Do declared local variable in innermost scope. That is for (int i=0; …)
- Do use += rather than + semantics to reduce the temporary variables.
- Don't create/destroy little objects in innermost loops; Java GC[3] slows thing.
- Don't use java.util.Vector in numerical computation.

Boisvert et al. [7] carried out series of SciMark [4] benchmark test on 500-MH Intel PIII running Win98. The results actually showed that Java (Sun 1.2 and IBM 1.1.8) outperforms C (Borland 5.5 and MS VC++ 5.0) with optimization. Java's performance is correlated to JVM implementation rather than underlying hardware [7].

2.5 J2ME: Real-world performance

Yi et al. [10] performed series of benchmark test on various PDA and wireless devices with J2ME. Each device is loaded with CLDC 1.0 and MIDP 1.0. The benchmark test includes: JKernelMark, JAppsMark and JXMLMark. JKernelMark is set of test drivers for testing KVM implementation while JAppsMark and JXMLMark are for applications. The JKernelMark benchmark includes basic numerical computation, string manipulation, memory management, and method calls. The benchmark test results can be found at:
http://www.javaworld.com/javaworld/jw-10-2002/images/jw-1025-j2mebenchmark4.gif

[3] Garbage Collector
[4] SciMark is benchmark from National Institute of Standards and Technology, http://math.nist.gov/SciMark.

Different JVM implementations actually give rather wide range of performances.

2.6 Garbage Collection in Embedded System

Chent et al. [11] performed set of experiment on relationship between garbage collection and energy consumption on Palm OS device. KVM uses mark and sweep style garbage collection algorithm. Overall the experiment showed that frequent garbage collection actually consumes less energy while it may impact application performance.

3 Project Plan

The purpose of the project is to compare runtime performance of Java and C on Palm OS device. Many of the previous Java and C/C++ works were performed either on Unix or Window machines where processor speed, memory and power are plentiful. One of the main challenges of this project is getting complicated algorithm programs running on the low power, stringent physical memory and limited processor speed PDA device. Three questions should be answered by end of this project: which language has better runtime performance on embedded PDA device, Java or C? If Java's performance is poorer than C on PDA device, how bad is it? Is there any future improvement could be made either on the JVM itself or the software written in Java? Java has many useful features that ease the programmer's responsibility to produce safe and robust software. Sometimes these useful features have to be sacrificed in order to boost up Java performance. For instance Java array and garbage collection are useful but also incurring huge runtime overhead.

3.1 Target test environment

Benchmark test is carried out on Sony Clie with Palm OS 4.0 and 16MB of physical memory. Sun Microsystems KVM is used as JVM.

3.2 Development environment

Development is carried out under Intel PIII 700-MH Win98. Require Java software's are JDK1.4 and J2ME (CLDC 1.0 and MLDP 1.0). Require C software's are Cygwin emulator with GCC and PRC-TOOL [14] for Palm OS.

3.3 Benchmark test

The performance measurement is based on the execution time and memory usage. Below are list of benchmark test programs that should be performed on the Sony Clie[5]:

[5] Eventual list may vary little depending on actual implementation and time constraints.

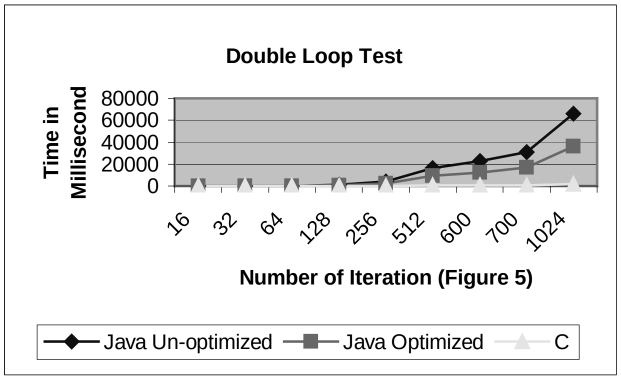
- Looping
- Array Copy
- Hashing
- String Concatenation
- Matrix Operation
- Factorial (recursive and looping)

Neither Java nor C can claim to be the only best language for embedded environment development. Java and C each has its advantages and disadvantages. Java's rich set of library and its runtime checking make development much faster and produce robust software while suffering performance issues. C on the other hand relies more on the developer's coding skills and language knowledge such as manual allocation and de-allocation of memory where development takes much longer and produce error-prone software while honoring with its excellent performance. There are trade offs on using either Java or C. It is up to developer deciding which language will benefit the most.

4 Results and Discussion

4.1 Looping

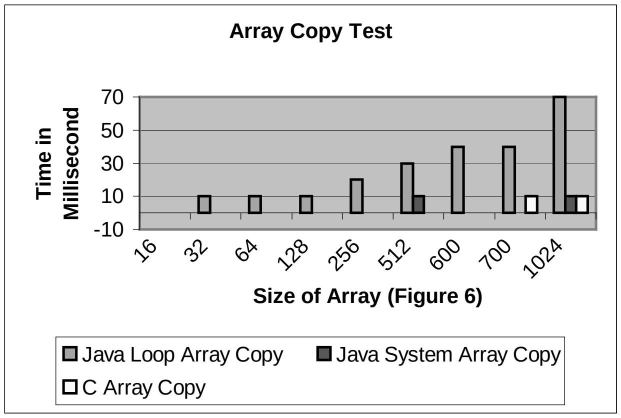
The experiment involves running calculations through double *for* loops with *n* numbers of iteration per loop. The execution time of C obviously outperforms Java (Figure 5).



Java must avoid putting unnecessary method calls and array access in the loop to improve the runtime performance. Array access involves automatic array bound check that can slow down array access. In the Optimized version of the Java loop test, array access is taken out of the loop, and the execution time shows great improvement.

4.2 Array Copy

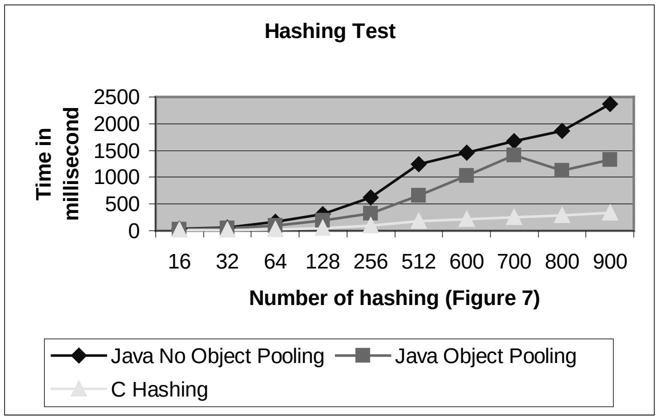
This experiment involves copy source array to destination array. Using loop to copy arrays in Java outputs poor performance. Shirazi [15] suggests using the 'system.arraycopy' to improve array copy execution time.



From figure 6, it is clear that using Java's system array copy API significantly improves the execution time and yields compatible results as C. In fact, for array size of 700 Java actually outperforms C by more than 10 milliseconds.

4.3 Hashing

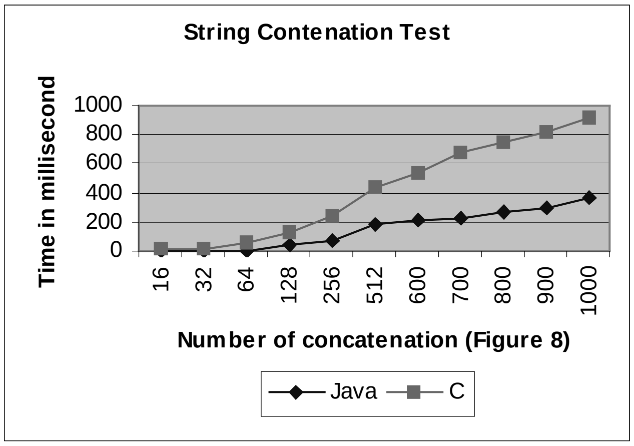
In Sosnoski's Java and C benchmark performance experiment, he points out the importance of Java object management [4]. Smart allocation and garbage collection of Java object will yield much better performance than just blindly allocation of unnecessary objects. Since garbage collection is an expensive operation, it should be avoided whenever possible. Object pooling is the re-use of Java object to avoid garbage collection while running heavy computation. In the *Object pooling version of Java hashing*, most frequent used objects are re-used without being garbage collected, thus the running time is very compatible to C. After all Java's build-in hashing library *Hashtable* frees programmer from re-implementing hashing routines. In the experiment C hashing is implemented using separate chaining algorithm[6].



[6] Uses Mark Weiss C hashing implementation http://www.niksula.cs.hut.fi/~tik76122/dsaa_c2e/files.htm

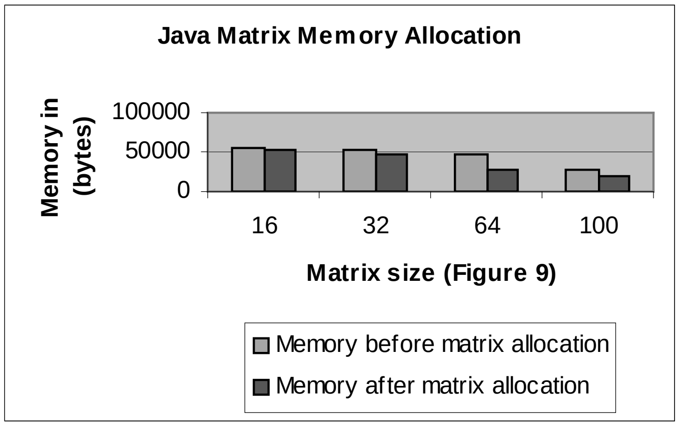
4.4 String Concatenation

Java's String class is easy to use and it comes very handy in all sort of situation. Concatenation of Java static strings is done at the compilation time, thus it takes off the runtime burden. C's string is implemented using array of characters or characters pointer, thus all string concatenation is done at the runtime. Java's string concatenation clearly out performs C's primitive string manipulation.

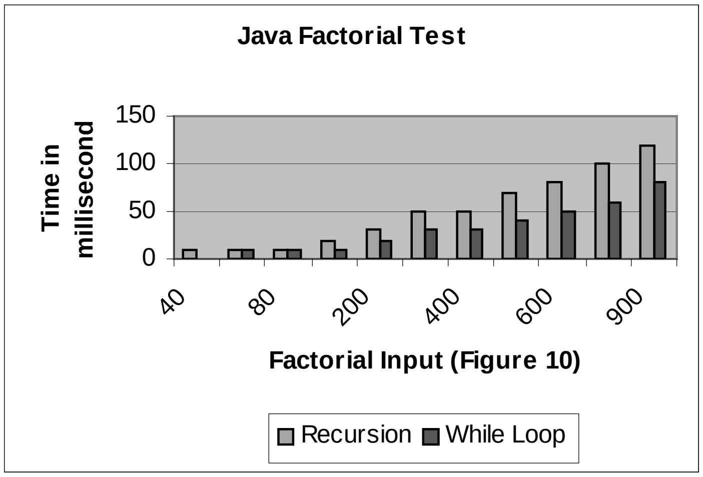


4.5 Matrix Operation

Memory shortage is the main problem with Java development in the PDA environment. PalmOS KVM only allocates 64KB of total memory for Java program to run. Matrix operation usually involves heavy memory usage and computation cycles. Matrix computation in Java is very poor due to the fact that matrix implementation is using multi-dimensional Java *array* in multiple *for* loops. Frequently garbage collection and array bound checks within loops are the main problems for Java matrix operation. There is a way to disable Java array bound checking, but KVM and JDK do not support this feature. GCC Java compiler can be tailored to meet such needs. During the experiment Java throws *OutOfMemory* exception when trying to allocate 128x128 integer type matrix. This behavior does concur with the 64KB memory limitation of KVM under Palm OS device. An interesting observation is that Java garbage collection is not very consistent during the execution of Java program. In figure 9, after memory allocation of size 64x64 matrix, the garbage collector does not seemed to be run, since the *Memory before matrix allocation* of size 100x100 matrix still remains to be around 28KB. Overall KVM under Palm OS is not an ideal place for matrix computation.



4.6 Factorial



Factorial is a highly recursive algorithm but it can also be implemented using *while loop*. The recursive version of programs in both Java and C are much shorter than the *while loop* version. From figure 10, it is clear that the *while loop* version of Java factorial runs significantly faster than the recursive version. According to Chirazi [15] Java should avoid recursion when ever possible instead loop should be used. C's recursive factorial throws a *stack over flow error* when input to factorial is 100. Palm OS clearly has a relative smaller stack size of 2.5KB. Thus Java seems to be doing better job in the factorial calculation.

5. Conclusion

In this report the performances of Java and C are closely studied on the Palm OS PDA device, Sony Clie. No single language is perfect for development on the PDA device. Both Java and C have to deal with the following issues:

- Memory management
  - o Java's automatic garbage collection helps developer catch memory leak by de-allocating out of scope objects,

but it can also cause significant performance issue, especially within loops.

o Smart object re-use and pooling in Java can significantly improve the memory allocation time and execution time, it prevents unnecessary object allocation and garbage collection.

o C relies on developer to manage the malloc and free of memory objects, which can cause subtle memory leak bugs.

o C needs better basic String operations. Operation such as StrCopy and StrCat are too error prone and slow.

- Execution time

o Java array access has poor performance due to the automatic array bound checking, but it can be disabled in GCC Java[7]. On other hand array bound check can be a very safety feature to eliminate runtime bug or memory violation.

o Java's loop execution time can be improved by taking out array access operation or other unnecessary method calls from the loops.

o Avoid recursion whenever possible can improve the performance of both Java and C.

o C's looping or matrix operations are fast.

- Development time

o Java's rich set of APIs definitely eases the software development on the PDA devices. Automatic garbage collection and array checking definitely take over much of the developer's responsibilities.

o C is too cumbersome to use and lack of good String and Array operation libraries on Palm OS devices.

With careful tuning and smart object management, Java's performance can definitely match up to C or even exceeding it.

6 Future Works

Java has a rich set of IO and network APIs that are interesting to look into on the PDA devices, since the future of PDA devices all have to be connected to the Internet and distributed. The performance of Java IO and network can be compared against C's system level IO and socket library.

References

[1] John W. Muchow. *Core J2ME Technology & MIDP*. Prentice Hall PTR, Saddle River, NJ, 2002.

[2] Sumi Helal. Standard, Tools, & Best Practices. *Pervasive Computing, ACM,* January, 2002.

[3] *Java 2 Platform Micro Edition (J2ME^{TM}) Technology for Creating Mobile Device*. White paper. Sun Microsystems, May 19, 2000.

[4] Dennis M. Sosnoski. Java performance programming, Part 1: Smart object-management saves the day. Java World, November 1999. http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html

[5] Dennis M. Sosnoski. Java Performance Comparison with C/C++. Sosnoski Software Solution, Inc. This report was presented in 1999 JavaOne conference. http://www.sosnoski.com/Java/Compare.html

[6] J. E. Moreira, S. P. Midkiff, and M. Gupta. A Comparison of Java, C/C++, and FORTRAN for Numerical Computing. IEEE Antennas and Propagation Magazine, Vol. 40, No. 5, October 1998.

[7] Ronald F. Boisvert, Jose Moreira, Michael Philipensen, Roldan Pozo. Java and Numerical Computing. *Computing in Science & Engineering*. March, 2001.

[8] Java Grande Forum. Improve Java Performance. http://www.javagrande.org

[9] SciMark. Java benchmark by NIST. http://math.nist.gov/SciMark

[10] Wang Yi, C. Y. Reddy, Gavin Ang. J2ME device: Real-world performance. Java World. March 25, 2002. http://www.javaworld.com/javaworld/jw- 10-2002/jw-1025-j2mebenchmark.html

[11] G. Chent, R. Shetty, M. Kandemirt, N. Vijaykrishnant, M. J. Irwint, and M. Wolczkot. Tuning Garbage Collection in an Embedded Java Environment. In Proc. Eight International Symposium on High-Performance Computer Architecture. 2002

[12] More detail on Java and C benchmark. http://www.spec.org

[13] David A. Cargill, Mohammad Radaideh. A Practitioner Report on the Evaluation of the Performance of The C, C++ and Java Compilers on the OS/390 Platform.

[14] Neil Rhodes, Julie McKeeban. Palm OS Programming. The Developer's Guide, 2nd Edition. O'Reilly & Associates Inc. 2002.

[15] Jack Shirazi. Java Performance Tuning. O'Reilly & Associates Inc. 2000.

[7] Time limitation does not allow to look into GCC Java