A Performance Analysis of Java and C

Ambika PAJJURI and Haseeb AHMED

***Abstract* – Java and C are two popular specification languages used to define systems of all sizes and forms. In this paper, we present a performance comparison of various algorithms written in C and Java on Windows NT and Linux environments. The metrics considered in the analysis include speed of execution, memory usage, Java vs. C overheads and other special features that characterize the two languages. We investigated both languages based on how their design choices influence their performance rather than by semantics and programming paradigms. The algorithms for the analysis were chosen to represent those commonly used in embedded systems (such as FIRs) as well as more exotic ones like the MD5 cipher. Our results show that, in general, C produced better run time performance than Java across both Linux and Windows NT platforms.**

***Index Terms*-Java, C, Performance comparison**

I. Introduction

THE design of a computer language often results from a desire to solve a set of problems in a given domain. Most modern languages strive to be the 'one size fits all' type of solution implying a broad set of goals. These often-divergent goals often lead to a 'specialization' effect wherein certain features are readily adopted into the mainstream and others fade away. We compare two such languages - Java and C.

Conventional wisdom suggests that Java and C make an odd pair to investigate. They do not share a common programming paradigm (object oriented vs. procedural). Moreover, Java tries to insulate users from the underlying architecture, while C is very accommodating to low-level access. It is perhaps for this reason that much of the published research work has focused on more natural comparisons such as Java and C++.

Java and C are both specification languages. C was conceived as a 'high level assembly' language whereas Java had its roots as an embedded/portable language for set-top boxes. The C language derived much of its semantics from its ancestor B, and so a simple procedural pass-by-value methodology was adopted. Java, due to its (very lucrative) requirement for portability and ease of use, chose to go with an object-oriented model. So while Java's internals grew to be more complex, the programmer was largely insulated from all the details.

Both Java and C have design choices that were intended to aid the programmer and (or) the compiler. Many of these features remain unused or unimplemented despite underlying hardware support. For example, hardware often has support for execution of MAC type instructions but there is no direct syntax for doing so in C or Java.

C allows a lot of flexibility to the programmer, but it is left largely to programmers and compilers to exploit these features. In the case of Java, the Java Virtual Machine (JVM), on which all Java programs run, hides many of the optimizations. Java, in its current form, is not very suitable for use in embedded systems. This is because does not support operations like direct memory access, interrupt handling and scheduling to meet hard deadlines.

The rest of this paper is organized as follows – In Section II, we present a summary of related work in this field. In Section IV, we discuss our project plan.

II. Related Work

A. The Java Performance Report – Osvaldo Pinali Doederlein

The Java Performance report [1] compares the performance of C and Java algorithms on Win32 platforms. The tests used a suite of algorithms written in C (BYTEmark) and their direct port to Java (JBYTEmark). The results presented in the paper indicate that, in general, C outperformed Java, as one would expect. However, the performance of Java depended on the underlying JVM, and also the specific algorithm under test. In fact, in some algorithms, specific Java implementations (especially IBM's JDK 1.3) outperformed C.

B. Binaries vs. Bytecodes - Chris Rijk

The results from Pinaldi's Java performance report [1] were further strengthened by Rijk's results [2] where IBM's JDK v1.3.0 was seen to outperform even Microsoft's Visual C compiler in many of the benchmarks, as shown in Fig.1. Some of the algorithms used were "Game of life" (an advanced implementation of J.H. Conway's simple cellular automaton), Fibonacci and FFT. This challenges the notion that the JVM is always an extra piece of luggage.

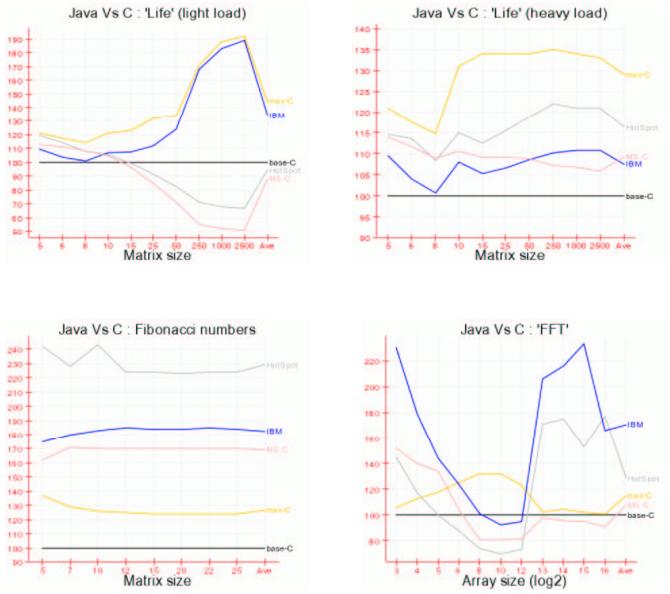

Figure 1 - Comparison results between IBM's JVM and C. From [2] Chris Rijk, Binaries Vs Byte-Codes. The ordinate for the plots is Mflops

C. *The Java Performance Analysis for Scientific Computing – Roldan Pozo*

In contrast to Pinaldi's report [1], Pozo [3] considered a more diversified array of algorithms commonly used in scientific computing. His approach was unique in that he worked with operations that were both CPU and memory intensive (*e.g.*, large matrix (1000x1000) multiplication operations). His observations were as follows:

C's strengths:

- Allows for direct mapping to hardware
- Provides more opportunities for optimizations
- No penalty for garbage collection

Java's strengths:

- Performance varies widely by the choice of a JVM – the best results were from IBM and Sun.
- Performance closely linked to underlying hardware (*i.e.* faster CPU does make an impact)

Pozo's experiments also showed that unlike the performance of C/C++ compilers, there is a lot of variation in the performance of the different JVMs. The application of some small non-standard optimization also produced significant benefits (as shown in Fig. 2). Considering the benefits incurred, such optimization should probably become the norm.

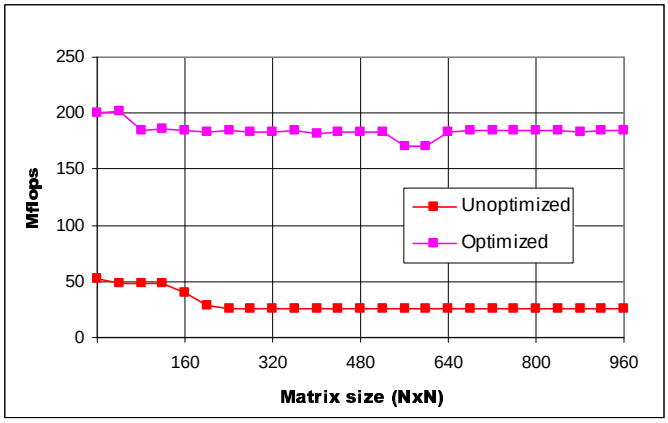

Figure 2 - Results of matrix multiplication from R. Pozo [3] showing that select matrix optimizations can yield significant improvements

R. Pozo concluded with two important comments:

i. Java requires more aggressive memory mechanisms to compensate for the gawkiness of automatic garbage collection. (This point is reinforced by Pozo's work [3]).

ii. JVMs are increasingly important in byte-code manipulation. (Also see for more recent research by Kyle [6] and Radhakrishnan [7]).

Other more subtle issues alluded to why Java was less favorable than C for use in large scientific and engineering applications. These include the lack of efficient multidimensional arrays, the inability to take advantage of fused multiply-add and associativity operations in compiler optimizations (also confirmed by Midkiff's result [8]).

D. *The Java Real-time Extension Specification*

Another emerging area for study is the Java 'Real-time Extension Specification' [5]. It is expected to bring long desired advantages of the Java Platform, like binary portability, dynamic code loading, tool support, safety, security, and simplicity, to an important industry segment: real-time systems. This extension targets both "hard real-time" and "soft real-time" systems. The specification addresses many issues, including garbage collection semantics, synchronization, thread scheduling, JVM-RTOS interface, and high-resolution time management.

III. Methodology and Goals

A. *Methodology*

We ran experiments over a gamut of algorithms written in both Java and C, on both Windows NT and Linux platforms. Each of the algorithms was run for different number of the appropriate variable or number of iterations, giving a larger data set for analysis. Moreover, each sample point was averaged over five runs to minimize random errors. All the algorithms were run using the same hardware to enable a meaningful performance comparison. Some of the programs were used from existing benchmarks and others written by us for the purpose of this comparison.

Details of the machine and compilers used in the experiments are given in Table 1:

Processor	Intel Pentium II 200 MMX
Memory	32MB
Linux OS	Kernel 2.2.12-20
WinNT OS	NT 4.00.1381 with Service Pack 6
Compiler - Java	java 1.3.1_01, Java HotSpot(TM) Client VM (build 1.3.1_01, mixed mode)
Compiler - C	gcc 2.95.3-5

Table 1 Details of hardware, operating systems and compilers used

Run times for Java were calculated as a combination of Java's System.currentTimeMillis() and computing the time before and after the run algorithm. Memory for Java was observed using Java's Runtime.getFreeMemory(). – We believe this is more accurate as it gives the algorithm run times and memory usage computing time and memory for only the relevant objects. We also observed the Task Manager for Windows NT and the Top monitor on Linux, to ensure that any garbage collection runs do not affect our results and that both results behave similarly.Runtimes and Memory for Java were obtained using Top on Cygwin for Windows NT and Linux.

To ensure a fair comparison, we chose algorithms that were not dependant on System calls to minimize kernel/library calls. Our run times for the C version of the tests show very little System time.

B. *Operating Systems and target metrics*

Linux and Windows have distinct architectures. This extenuates C and Java's design where C likes to be close to the native OS while Java relies on its JVM. In considering our metrics for evaluation some of the factors influencing the algorithms chosen and the tests run include:

i. Memory management is one of the key differences between Java and C. We tried to expand on Milo Martin's [4] work to identify other such opportunities for enhancements to both Java and C.
ii. Run times for the two versions. With C being close to the OS, it is expected to work much faster than Java. We tried to see how well Java performed with respect to C.
iii. Another area that has not been well investigated is the primitive data type selection in Java. Strings in particular pose a challenge because they consist of 16-bit Unicode. We ran some tests on String concatenation to see how they fare in Java vs. their C counterparts.
iv. We tried to analyze the numbers for simple DSP operations to see if Java has a future in Real time systems.

C. *Algorithms*

The suite of algorithms we used was both CPU and memory intensive. The algorithms that we ran are shown in Table 1 below –

Algorithm	Characteristics
Basic FIR	Traditional DSP 'multiply then add' computation
Matrix Multiply	Exercise memory and CPU
MessageDigest5 (MD5)	32bit-CPU friendly.
Ackermann's	Highly recursive algorithm.
Fibonacci Series	Recursion with ADD operations
Simple Hash	Memory traversal
Array Copy	Exercise mem-to-mem operations
String Concatenation	Unique to Java and C.

Table 2 List of algorithms used

In this paper, a subset of the results is presented. We have chosen the subset based on uniqueness of results or if some unexpected or anomalous results were observed. An Excel spreadsheet with all the results can be downloaded from http://www.columbia.edu/~ap714/COMS4995/results.xls

IV. Results and Discussion

1) *FIR and MD5*

The run time results for MD5 and FIR for the four test configurations are shown in Figure 3. The results show that the run times for Java are compatible with C. The results seem to imply that Java's performance is good enough for at least some DSP type applications.

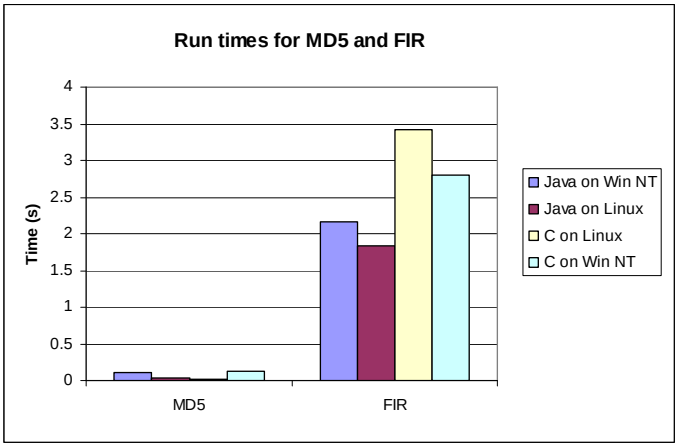

Figure 3 - Run time results for MD5 and FIR.

However, it is to be noted that timing alone is not sufficient for use on Embedded Systems. A typical JRE on Window's requires over 40MB of free space before attempting to install it. On Linux, it requires about 45 megabytes of free disk. In addition it requires a minimum RAM of 32MB. C is close to the native OS, while Java is built on the JVM. The following figure gives a high level view of how differently applications in Java and C operate.

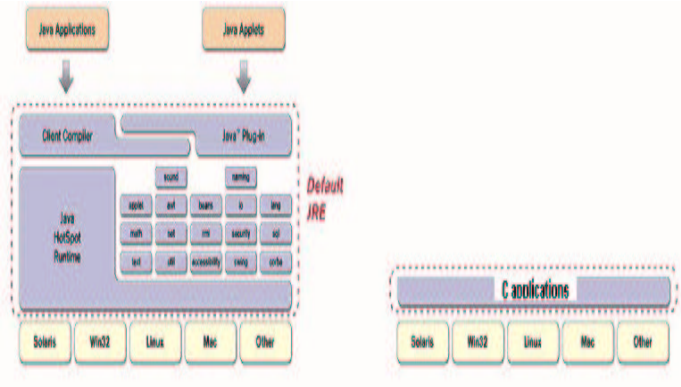

Figure 4 - JRE and C on the different operating systems

Newer stripped down JVMs are aimed at using lesser memory and providing good predictable runtimes. For Example, the Java Embedded Server 2.0 software, including all the services required by the OSGi standard, requires 900 KB of persistent memory and 2.1 MB of DRAM. Java Embedded Server 2.0 has been tested on Solaris[tm], Microsoft Windows NT 4.0, Linux (JDK 1.2), and VxWorks

(PersonalJava[tm] 3.0.2) platforms. The small size, architecture, use of Java technology, and extensibility are targeted primarily at developers and manufacturers of network-enabled products. Examples of these targeted applications are PDAs, cell and web phones, set-top boxes and televisions, medical and industrial devices, gauges and meters, ATMs, gas pumps, manufacturing equipment, telecommunications equipment and devices, and network communications equipment such as routers and switches. It remains to be seen if the potential would be realized soon.

2) FibonacciSeries

The run time results for Fibonacci series for all the configurations are shown in Figure 5. The results showed, surprisingly, that Java performed better than C, although very marginally. This runs counter to the myth that C code is always faster than Java.

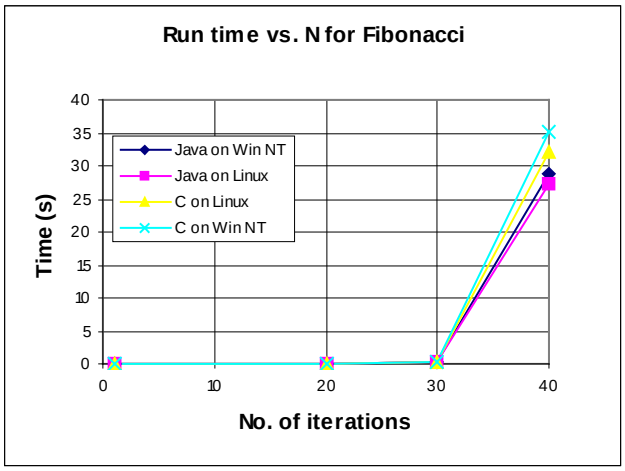

Figure 5 - Results for run times of Fibonacci series fare better for Java than C.

This effect can be explained in part by the recursive operations that require continual stack management. The Fibonacci assembly code snippet in the appendix explains this further. Even though the test program was solely devoted to running the Fibonacci program, there is still poor register allocation with penalties for restoring the stack. Java bytecode (see appendix) revealed additional register use to accommodate this situation. The assembly code shows also that branches are followed by multiple *nop* statements – Contrast this to an extremely recursive algorithm like Ackermann. Java's attempts to balance register allocations not knowing that the stack will be frequently updated adds a lot more overhead that ultimately degrades performance significantly.

3) Array copy operation

Both run times and memory usage was higher for Java than for C. The run times for both C and Java were approximately constant for array sizes between 10 and 10000. The run time increased dramatically for larger array sizes.

4) Matrix operations

Matrix operations again gave better runs for C than Java. An interesting point discussed earlier in this paper is the effect of bounds checking. While this does incur extra operations and time, it is a very good safety feature. A feature considered in the original Java proposal called "asserts" is an alternative to this. It gives a hint to the compiler and skips the bounds checking. This feature should probably be re-introduced in Java.

In addition, we also encountered a java.lang.OutOfMemory error with matrix size of 3000X3000. This is due to the heap size of the VM. While the heap size can be increased using VM options -Xmx (should be larger than the setting for –Xms). What this does imply is that such large operations should be performed piecewise.

5) Ackermann

The results obtained for memory usage for Ackermann algorithm are shown in Figure 6. The Ackermann[n] algorithm is a massively recursive algorithm. The interesting result of this experiment (in addition to a comparison of the memory usage) is that the Java Virtual Machine crashed on both the Windows and Linux platforms for N=20 and N~370000 respectively. The error generated in both cases was a "Stack Overflow" exception. The JVM has a maximum stack size that is configured at the VM compile time. While there are command line options (namely -Xss) that make the stack size configurable, our experimentation leads us to believe that this functionality does not work. (Same experience reported by developers on the Sun Microsystem Java developer website) The only alternative is that for highly recursive functions, care should be taken to do the job piecewise just as is done for large matrix multiplications.

C on the other hand, seemed to develop a voracious appetite for more and more memory during this test. At one point about 4K was being consumed every 2 seconds! This was anticipated, as the stack is central to C's 'pass by value' design. Program execution speed also deteriorated due to the intensive stack management.

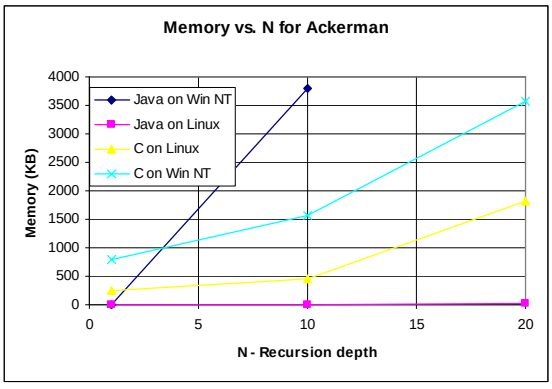

Figure 6 - Memory usage for Ackerman. The Java on NT crashed for N=20

6) Hash

Java performs well for simple hash functions. However, with somewhat more complex hashing, the memory usage increased very quickly for Java. The results for complex hash functions are shown in Figure 7. In addition, we noticed some anomalous dips in memory usage for increasing number of iterations (over multiple runs). This may be due to the garbage collector.

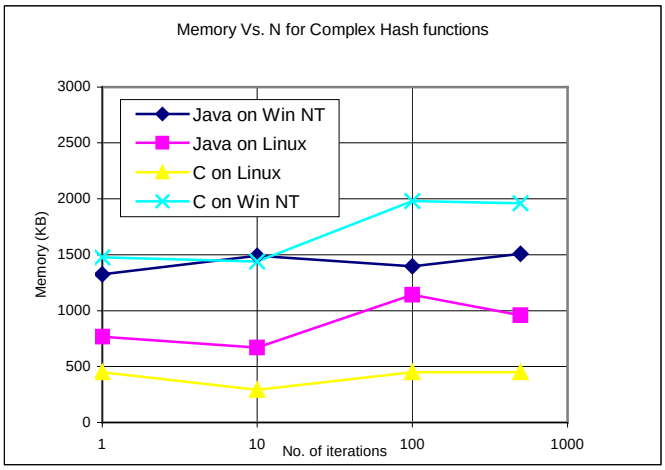

Figure 7 - Memory usage for different iterations of a Complex hash function.

7) String concatenation

Java's String class has been designed differently from other classes. The String class is Immutable and concatenating strings creates multiple intermediate representations. StringBuffers are used by the compiler to implement the binary string concatenation operator '+'. We used the StringBuffer's append method for concatenation in our tests. The memory usage results obtained for string concatenation are shown in Figure 8.

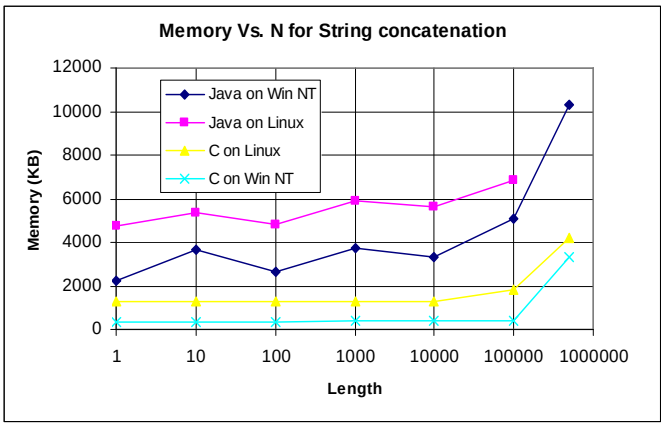

Figure 8 - Results of Memory Vs. N for String Concatenation operations. C performs better than Java

As in the complex hash case, we noticed some kinks with anomalous dips in memory usage with increasing length.

Every string buffer has a certain capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger. Due this string buffer structure, it is possible that occasional reductions in memory take place.

Another point to note, is that the StringBuffer's append method is synchronized. While this is important to support Java's multithreading, it is not inexpensive. Each call to the append method requires a lock on the StringBuffer object to be acquired and released. While this may be necessary to support multithreading, it may sometimes be overused and lead to slow and expensive code. In general care must be taken not to overdo synchronization in applications to avoid sluggish performance.

V. Analysis

In this section we present the overall analysis of the key metrics from our results.

A. Execution Speed

This factor is readily visible to programmers (and end users). Execution speed is often seen as the most important attribute of a language's performance. As for this metric, run times were better for C than Java, but Java's performance was not bad particularly when we look at the FIR and MD5 results.

B. Memory Usage

Is this a moot point in these times of cheap memory? We do not think so – especially since embedded systems have far more stringent memory requirements. Minimizing memory usage is becoming increasingly important, as expensive (and slow) I/O is still the bottleneck, even with faster CPUs. Will the smaller stripped off JVMs live up to the expectations? Time will tell.

We saw some rather unexpected results during some of the runs such as Hashing and String Concatenation with Java. These can be attributed to the unpredictable manner in which Garbage collection runs in Java and the creation/deletion of objects. The real time specification for Java, does address the unpredictable garbage collection issues.

C. Language Features

Java with its runtime optimizations, garbage collection and freebies like bounds checking seem very impressive. C on the other hand places the entire burden on the programmer. We tried to analyze the cost(s) and effects of some of these.

We have also tried to see how well Java's string concatenation behaves compared to similar operations in C. C does perform better, but the convenience of using this feature in Java is indisputable. Java's support for multithreading puts the onus of ensuring proper synchronization on the programmer. While using synchronization is important it can sometimes lead to sluggish and expensive runs.

In order to facilitate testing, a customized Java Front-End was written to try and capture several runs at once. This had the (unintentional) side effect of exposing the garbage collector in Java. Some of the behaviors observed were consistent with the mostly-concurrent Garbage Collector described in an earlier incarnation of Java (version 1.2_01) (see Printezis' report [9]). This proved to be a challenge because the garbage collector appears to employ a 'generational' model for deciding when objects must be automatically recalled. The effects were most visible when under some tests the free memory on the JVM would suddenly appear to grow after a test run that was intended to consume memory. Consequently, the execution time would be distorted, as it would now reflect the additional time consumed for garbage collection.

VI. Future Work

An interesting area for future work would be to perform some of the memory management experiments on Real time devices. This can be done once some kind of JVM is standardized and available for those devices. The Java Real Time Specification discussed in Hardin's article [5], tries to address issues important to real time devices. It would be particularly interesting to see if the Garbage collector does behave more predictably. Printezis and Detlefs' work [9] suggests that the interruptions caused by the garbage collector can be minimized by exploiting parallelism between marking and sweeping garbage. Our experience shows that algorithms that require multiple intermediate values should be first optimized away so that the garbage collector's task is kept as small as possible. Java has the advantage of optimizing at the bytecode level and C compilers already optimize away temporaries so there is certainly good precedence to pursue in Java. Another area of future investigation is the effect of Java's stack size on real-time specific applications.

Other areas for future work in the area include experiments on how the scheduler works and the effects of compiler optimizations. The scheduler can be enhanced to exploit the underlying processor. Byte code representation presents a nice way to utilize processors that are increasing RISC-CISC hybrids without having to rewrite source code again.

VII. Conclusions

Just as any one language cannot lay claim to solving all problem domains, our performance analysis rates Java and C on different metrics. We hope this will aid in the selection of the right language for the right task and provide future opportunities for exploration.As far as Java is concerned we think that the major challenge for its use in Embedded Systems are:

1. Memory
2. DMA
3. Unpredictable Garbage collection

With memory becoming cheaper and if the effort for a real time JVM is kept, Java will be very attractive for many non-critical applications. It is likely that it would be particularly popular for use on small devices.

References

[1] Osvaldo Pinali Doederlein, *The Java Performance Report - Part III*, http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/part3.html

[2] Chris Rijk, *Binaries Vs Byte-Codes, Ace's Hardware*, June 27, 2000.

[3] Roldan Pozo, *"Java Performance Analysis for Scientific Computing"* - National Institute of Standards and Technology, USA. This report was presented at the UKHEC: Java for High-end Computing in Nov 2000.

[4] Milo Martin, Manoj Plakal and Venkatesh Iyengar, *"Top-Level Data-Memory Hierarchy Performance: Java vs. C/C++"* - University of Wisconsin – Madison, Dec 1996.

[5] David Hardin, *"Bringing Java's benefits to real-time developers"* Dr. Dobb's Journal February 2000

[6] Kyle R. Bowers, David Kaeli, *"Characterizing the SPEC JVM98 Benchmarks on the Java Virtual Machine"* – Northeastern University Computer Architecture Research Group.

[7] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio and J. Sabarinathan, *"Java Runtime Systems: Characterization and Architectural Implications"*. A preliminary version of this paper appeared in the International Conference on High Performance Computers and Architectures (HPCA-6), Feb 2001.

[8] S.P. Midkiff, J.E. Moreira and M. Snir, *"Optimizing Array Reference Checking in JAVA programs"*, IBM Systems Journal, 37 (409-453) 1998.

[9] Tony Printezis and David Detlefs, *"A Generational Mostly-concurrent Garbage Collector"*, Sun Microsystems Technical Report Series, June 2000.

VIII. Appendix

1) Assembly code snippet for Fibonacci:

```
pushl  %ebp
movl   %esp,%ebp  // Normal program startup
pushl  %ebx
cmpl   $0x1,8(%ebp)
jle    0x2e <8048998>
movl   8(%ebp),%eax
decl   %eax // data passed on the stack
pushl  %eax
call   0xffffffec <fib>
addl   $0x4,%esp // stack restoration
movl   %eax,%ebx
movl   8(%ebp),%eax // Again data on stack
addl   $0xfffffffe,%eax
pushl  %eax
call   0xffffffd9 <fib>
addl   $0x4,%esp // stack restoration
movl   %eax,%eax
leal   (%eax,%ebx),%edx
movl   %edx,%eax
jmp    0xd <80489a0>
nop
jmp    0xa <80489a0>
nop
nop
movl   8(%ebp),%edx
movl   %edx,%eax
jmp    0x1 <80489a0>
nop
movl   -4(%ebp),%ebx
leave
ret
```

2) Bytecode for Fibonacci (using javap)

```
Compiled from fibo.java
public class fibo extends java.lang.Object {
   public fibo();
      /* Stack=1, Locals=1, Args_size=1 */
   public static void main(java.lang.String[]);
      /* Stack=2, Locals=2, Args_size=1 */
   public static int fib(int);
      /* Stack=3, Locals=1, Args_size=1 */
}

Method fibo()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return

Line numbers for method fibo()
  line 3: 0

Local variables for method fibo()
  fibo this  pc=0, length=5, slot=0

Method void main(java.lang.String[])
  0 aload_0
  1 iconst_0
  2 aaload
  3 invokestatic #2 <Method int
parseInt(java.lang.String)>
  6 istore_1
  7 getstatic #3 <Field java.io.PrintStream out>
 10 iload_1
 11 invokestatic #4 <Method int fib(int)>
 14 invokevirtual #5 <Method void println(int)>
 17 return

Line numbers for method void
main(java.lang.String[])
  line 5: 0
  line 6: 7
  line 7: 17

Local variables for method void
main(java.lang.String[])
  java.lang.String[] args  pc=0, length=18, slot=0
  int N  pc=7, length=11, slot=1

Method int fib(int)
  0 iload_0
  1 iconst_2
  2 if_icmpge 7
  5 iconst_1
  6 ireturn
  7 iload_0
  8 iconst_2
  9 isub
 10 invokestatic #4 <Method int fib(int)>
 13 iload_0
 14 iconst_1
 15 isub
 16 invokestatic #4 <Method int fib(int)>
 19 iadd
 20 ireturn

Line numbers for method int fib(int)
  line 9: 0
  line 10: 7

Local variables for method int fib(int)
  int n  pc=0, length=21, slot=0
```