Expanding the high performance embedded computing tool chest

- Mixing C and Java™

Nazario Irizarry, Jr
The MITRE Corporation
Bedford, MA

***Abstract*— High performance embedded computing systems are often implemented in the C language to achieve the utmost in speed. In light of continued budget reductions and the ever-present desire for quicker development timelines, safer and more productive languages need to be used as well. Java is often overlooked due to the perception that it is slow. Oracle Java 7 and C were compared to better understand their relative performance in single and multicore applications. Java performed as well as C in many of the tests. The quantitative findings and the conditions under which Java performs well help design solutions that exploit Java's code safety and productivity while yielding high performance mixed language solutions.**

Keywords—software performance; embedded software; high performance computing

I. INTRODUCTION

Developers of High Performance Embedded Computing (HPEC) software write code not just for computational processing but also for management and infrastructure tasks such as communication, processor/core allocation, task management, job scheduling, fault detection, fault handling, and logging. HPEC software has required extensive development to get the highest performance relative to limitations on size, weight, and power dissipation. In the past, these limitations have driven the choice between dedicated logic and programmable processors.

Significant improvements in processing capabilities, reduced feature sizes, and multiple processor cores have become available in commodity HPEC processor designs. Linux has become a commodity operating system with support for real-time scheduling, symmetric multiprocessing, and other features previously found in specialty operating systems. C, though known for speed, is not known for ease of development, flexibility, and code-safety. Java, which is known for higher productivity and code-safety, is not generally thought of for high performance.

It makes sense to consider utilizing both Java and C on a commodity processor with a commodity operating system for some HPEC designs. Understanding the relative performance tradeoffs of each is necessary for an effective design. The potential near-term benefits of a mixed application would be increased productivity, reduced hardware cost, reduced development time, enhanced overall code-safety, and potentially enhanced capability gained by leveraging the large repository of Java code, libraries, and tools. Long-term life-cycle benefits could also ensue as hardware and operating systems are upgraded since the Java applications are less sensitive to hardware and operating system details.

This paper examines the performance of Java and C on tasks that are typical of HPEC applications. It examines Java as a compute language. It looks at options that allow Java to invoke native libraries, especially for utilizing graphics processing units. If Java performs reasonably on these kinds of tasks then that opens the door to using Java as an adjunct to C in all kinds of tasks.

II. JAVA "WARM-UP"

Java performance varies with different vendors' implementations. This paper focuses on Oracle Java 7, Standard Edition (1.7.0_02-b13). Unlike C, that loads applications in "large chunks," Oracle Java 7 loads classes as needed at runtime -- requiring a lot of disk I/O. Classes must be found, validated, and their static values must be initialized. Initial execution is slow because initially code is interpreted.

Beyond initial execution, Oracle's Java Hotspot [1] monitors in-memory execution and adaptively optimizes frequently exercised code paths into the native instruction set. Method calls that can be safely in-lined [2, 3] can have their code moved into calling methods to avoid the overhead of context saving, stack manipulation, and method invocation. The net effect, shown in Figure 1, is that Java gets faster. Many HPEC applications run for hours or days and can tolerate a warm-up time.

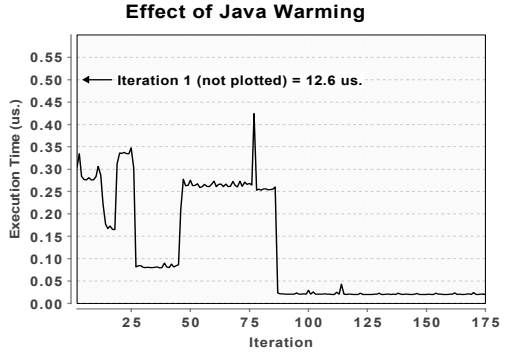

Fig. 1. Effect of Java Warming on Execution Time

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

Figure 1 is an example of how execution time decreases during the warm-up stage. The test executed a memory-based algorithm repeatedly. On each iteration, a red-black tree (described further below) was created and populated using a pseudo-random series of key-value pairs. The first iteration took 14 us. which is off-scale. Iterations 2 to 26 took from 0.16 to 0.3 microseconds. At iterations 27, 46, 94, and 301 this changed to 0.08, 0.22, 0.020, and 0.010 microseconds respectively. The final speed was much faster than the earlier speeds.

III. Java Computation Compared With C

The ultimate goal is to exploit the strengths of each language in a mixed application. However, the question arises as to whether Java can be used for computation by itself. Several numerical benchmarks were created in which algorithms were identically implemented in C and Java. Code that existed only in C was translated to Java and vice-versa. This was facilitated by similar language syntax for method definition, argument passing, and primitive-type variable declaration. The tests presented below were performed on Dell OptiPlex™ 960 systems. These feature a four-core Intel Core 2 Quad Q9650 @ 2.99 GHz (no hyper-threading [4]). The operating system was Linux with kernel version 2.6.32-220.17.1.el6.x86_64. The C code is compiled with option "-O3" using GCC version 4.4.6. No attempt was made to exploit vector machine instructions. Clock granularity was measured to be 34 nanoseconds. The sample-to-sample measurement jitter (using C code) was approximately 18 nanoseconds. All timing results for Java were collected after a suitable warm-up time – typically only a few seconds. Plot data were averaged from hundreds or thousands of trials.

A. Computation – Matrix Multiply

In this test a single array was allocated for the matrix coefficients. This avoided multi-dimension indirection. The coefficients were of type double. The matrices were square with dimensions ranging from 10 to 88. The matrix-multiply code consisted of three nested loops. The results are shown in Figure 2. It is surprising that the Java throughput (in giga-flops) was consistently a little higher than that of C. Over the range $40 \leqslant N \leqslant 60$ the Java compute throughput exceeded C by a typical three percent. Over the range $75 \leqslant N \leqslant 88$ this was about 10 percent. The characteristics of the matrix multiply kernel can be compared with those of the test in the next section. The characteristics are:

- Small code kernel,
- Loop bounds that are fixed or easily determined by the compiler,
- Array indexes with simple increments (increments are fixed within some of the loops),
- No memory allocation/de-allocation within the kernel

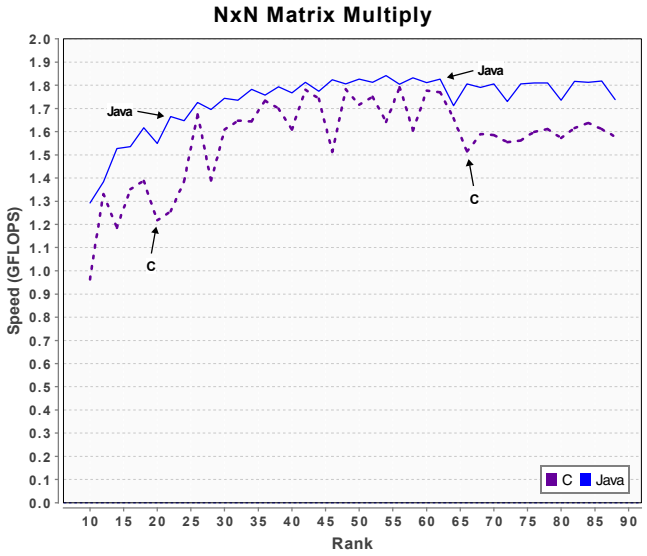

Fig. 2. Matrix Multiplication Speed

B. Computation – FFT

This test employed a fast Fourier transform (FFT) implementation by Columbia University [5]. The C version of the test was translated from the original Java version. The test used complex data, and implemented a non-optimized, in-place, radix-2 algorithm.

The results for data lengths from 26 to 218 are shown in Figure 3. Java performed at 60 to 85 percent of the speed of C. (An FFTW [6] implementation was tested for reference and was found to perform twice as fast as this C implementation.)

In contrast to the simple array-indexing scheme of the loops in the previous test, FFTs exhibit the following characteristics:

- The code kernel is longer,
- There are more array indexes used,
- Many array indexes are not incremented using fixed values which makes them more difficult to analyze at runtime,
- Inner loop bounds are more intricate functions of computed outer loop values
- There are more array lookups

It is hypothesized that "irregular" array indexing might be responsible for the inability to achieve a higher throughput for this algorithm. On simple loops Java's runtime array-bounds checking [7] can be done once before entry into a loop. More intricate indexing requires the bounds checking to be done for each array lookup and can add significantly to execution time. However, no further testing was performed to prove this hypothesis.

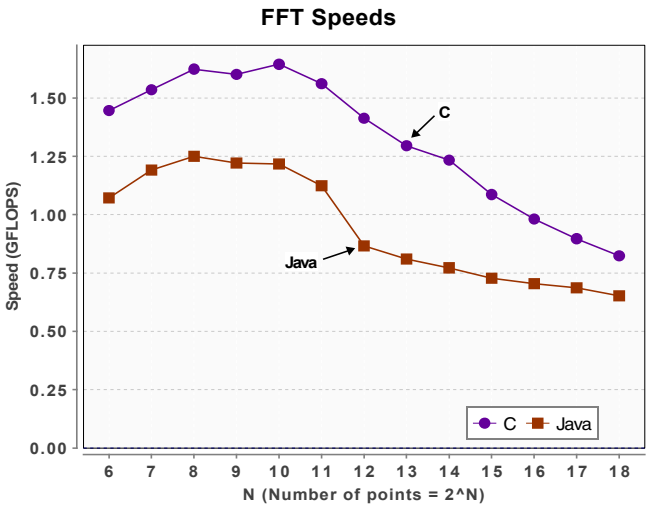

Fig. 3. FFT Speed

C. *Bit "Twiddling"*

Some central processing unit (CPU) instruction sets provide machine instructions to directly test and set individual bits in memory. Without resorting to those, bit-manipulation in Java or C is done by performing Boolean operations on bytes or integers in memory.

A benchmark was designed to test the performance of generating linear feedback shift-register [8] (LFSR) sequences using well-known polynomials [9]. On each iteration of the test, multiple sequences with polynomial orders ranging from 3 to 19 were generated. The array to hold the sequence was pre-allocated so that the benchmark timing was not affected by memory allocation. Table 1 shows that Java performed about twenty-eight percent faster than C on this combination of hardware, operating system, and code.

TABLE I. BIT MANIPULATION TIMES

C	2.8 msec.
Java	2.0 msec.

D. *In-Memory Data Structure Creation*

The previous tests used pre-allocated arrays. Here the goal was to test Java memory allocation, garbage collection (GC), structure creation, and in-memory traversal of intricate memory-based structures. A red-black tree [10] structure was chosen. These are binary trees that maintain sorted mappings of keys to values. As keys are added the old structure must be traversed to find the right place to insert new key-value pairs. Portions of the tree are regularly rebalanced to minimize the depth of the tree.

An existing C implementation of a red-black tree algorithm [11] was used and also translated to Java. In an insertion test pseudo-random integer keys were paired with "values" that were arrays with a size selected for each test. For each key-value pair the array value was allocated dynamically. The time to generate the pseudo-random key was not included in the results in Figure 4.

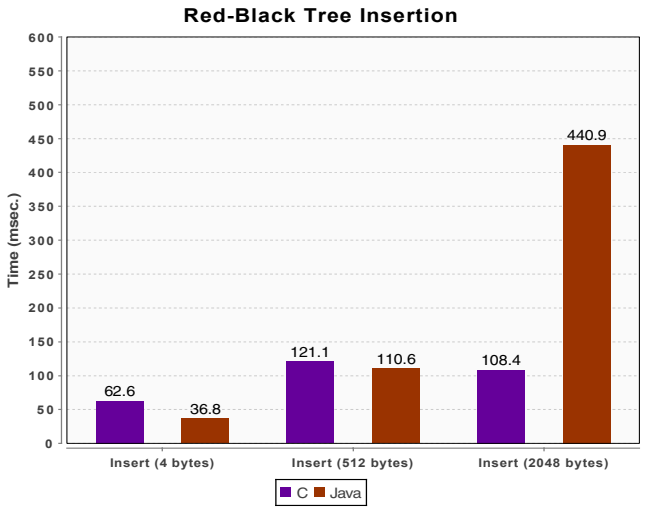

Fig. 4. Red-Black Tree Insertion

In Figure 4 the time to allocate and insert the key-value pairs into the tree, as well as the time to garbage-collect the tree was included in the results. Figure 4 shows that Java did quite well relative to C when four and 512-byte values were allocated. Figure 5 show the results of a lookup-only test. This shows traversal performance free of memory allocation/de-allocation. Here Java performed five percent worse for the four and 512-byte values, but 18 percent better than C for the 2048-byte values.

IV. JAVA OVERHEAD

One of Java's strengths is that it provides automatic garbage collection (GC), freeing the developer of the work and of the errors associated with freeing unused structures. However, this requires CPU time to execute. On multicore machines where the concurrent application workload is less than the number of cores, the GC can run concurrently with the compute threads on available cores. In these cases the effects of Java's concurrent garbage collectors [12, 13] are usually negligible.

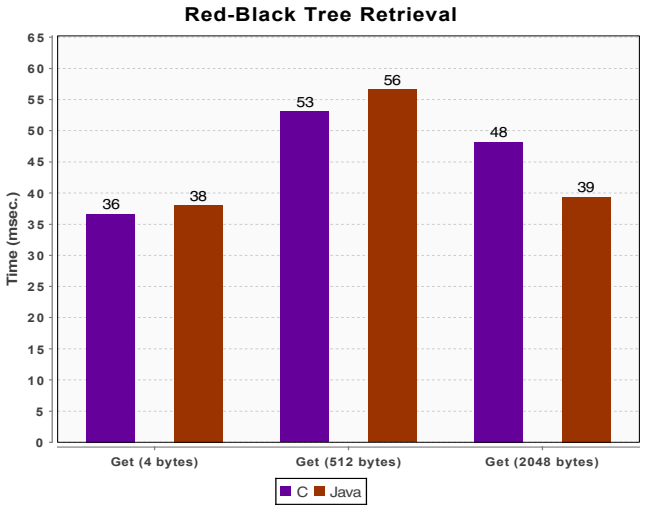

Fig. 5. Red-Black Tree Lookup

Two tests were used to measure the impact of the GC – one exercised the GC intensely, the other required no GC maintenance actions. Each test ran a compute workload in each of two Java threads. Each test was run with two and three cores allocated to the application. When only two cores were allocated the GC competed with the main work threads. When three cores were allocated the GC was free to run on the third thread without impact to the main work threads.

The red-black test described earlier was used for the memory intensive test. The application memory parameters were intentionally "poorly" set low to force frequent GC actions -- 7.5 times per second. The non-memory intensive test was based on the FFT algorithm described above. Its arrays were pre-allocated requiring no GC cleanup.

The results of these tests, shown in Table 2 provide a sense of the impact of GC in a poorly tuned application -- fifteen percent in this case. The results suggest that the best strategy for maintaining Java performance is to keep the garbage collector quiet by pre-allocating working buffers and objects when possible, minimizing dynamic memory activity and GC activity.

TABLE II. GC OVERHEAD

	Minimal Activity GC Test	High Activity* GC Test
3 Cores	155.7 sec.	155 sec.
2 Cores	155.0 sec	178 sec.
GC Impact	Negligible	15 percent

* 7.5 GC actions per second

V. USING GRAPHICAL PROCESSING UNITS

Utilizing graphics processing units (GPU) from Java requires a fast callout mechanism from Java to C that allows data to be passed efficiently and a Java API that provides access to the GPU via underlying C interfaces.

Five bridging libraries/tools were investigated – JNI [14], JNA [15], BridJ [16], SWIG [17], and HawtJNI [18]. JNI provided the best performance but was the most tedious to use. BridJ and SWIG provided the next best performance. A comprehensive comparison of BridJ and SWIG would be lengthy as there are various factors to consider such as the number of method arguments, whether data are the Java heap or native, whether the data are used in a read-only or read-write manner or just managed by Java and not accessed by Java at all. Table 3 shows results for a small combination of these variables. One is not consistently better than the other. Furthermore, BridJ and SWIG differed in ease of use, the amount of development to generate proxy classes, and the amount of work to build an integrated application. The best strategy was to allocate working buffers natively to avoid the overhead of exposing Java array data to C. BridJ made it easy to use Java to manage native data (including garbage collection), and to invoke C to process it.

TABLE III. BRIEF COMPARISON OF BEST PERFORMING JAVA-TO-C BRIDGING TOOLS

Call Arguments	JNI	SWIG	Bridj
2 integer args.	0.11 ns.	0.15 ns.	0.11 ns.
4 integer args.	0.11 ns.	0.17 ns.	2.0 ns.
2 read-only Java arrays*	13 ms.	190 ms.	110 ms.
Pointers to 2 native arrays* referenced by Java proxies	N/A	0.58 ns.	0.1 ns.

[a.] Double arrays of length 200

JavaCL [19] is a Java library that provides access to GPU's that support OpenCL [20]. JavaCL utilizes BridJ. A benchmark doing FFT computation on GPUs was developed. It used optimized FFT kernels developed by Apple Inc. [21]. Apple's original C code was used and also translated as directly as possible into Java using BridJ for native memory allocation and the JavaCL library for OpenCL access.

Figure 6 shows the FFT throughput that was achieved as a function of data size. Measurements were made for both synchronous and asynchronous GPU invocation. Using native data buffers allowed the Java version to perform almost as well as the C version.

Another performance metric is how much time it takes to set up arguments and enqueue an OpenCL kernel invocation. Though the GPU's were being kept busy, Table 4 shows that there was an increased kernel invocation time of approximately 2 us. Since the Java library must utilize the underlying C native interface the 2 us. is the additional Java load.

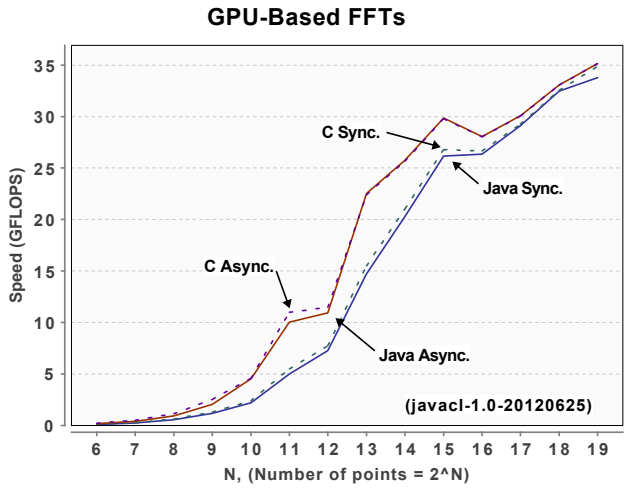

Fig. 6. Throughput of GPU-Based FFTs

TABLE IV. GPU INVOCATION TIME

C	9.1 us.
Java	11 us.

VI. CONCLUSIONS

Java performs so well after warm-up that it should be used to develop intricate mixed language Java/C HPEC applications. The Java garbage collector detracts from performance if there is a significant amount of dynamic memory allocation and de-allocation. On the other hand the practice of pre-allocating data buffers and structures at application start applies as well to Java as it does to C. In a mixed application it is likely that Java would be used to manage data and perform infrastructure tasks while C would likely be used for numerical processing. BridJ and SWIG do well at bridging Java-to-native callouts. The HPEC developer will also be interested in how Java frameworks can facilitate multi-node, non-shared-memory communication and processing. However, that remains a question for a future investigation.

ACKNOWLEDGMENT

Thank you to Paul Silvey and John Gibson for help reviewing this paper. Thank you to Brian Sroka and David Koester for being internal consultants on this work. The benchmarks and timing tools used in this study are available at [22] and [23]. A tool to generate code to bridge calls from C to Java is available at [24].

REFERENCES

[1] "Java SE HotSpot at a Glance." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>

[2] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2001. A dynamic optimization framework for a Java just-in-time compiler. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01). ACM, New York, NY, USA, 180-195. DOI=10.1145/504282.504296 <http://doi.acm.org/10.1145/504282.504296 >

[3] "The Java HotSpot Performance Engine Architecture." Oracle Technology Network. <http://www.oracle.com/technetwork/java/whitepaper-135217.html#method>

[4] "Intel Hyper-Threading Technology: Your Questions Answered." Intel Corp. <http://software.intel.com/en-us/articles/intel-hyper-threading-technology-your-questions-answered>

[5] "FFT.java." MEAPsoft/Columbia University. <http://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT_8java-source.html>

[6] "FFTW." <http://www.fftw.org/>

[7] Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck. "Array bounds check elimination for the Java HotSpot™ client compiler." Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 125–133. PPPJ 2007, September 5–7, 2007, Lisboa, Portugal. <http://ssw.jku.at/Research/Papers/Wuerthinger07/Wuerthinger07.pdf>

[8] Solomon W. Golomb. 1981. Shift Register Sequences. Aegean Park Press, Laguna Hills, CA, USA. ISBN:0894120484

[9] "Linear Feedback Shift Registers." New Wave Instruments. <http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm>

[10] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009). Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

[11] Emin Martinian, "Red-Black Tree C Code." <http://web.mit.edu/~emin/www.old/source_code/red_black_tree/index.html>

[12] "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>

[13] "The Garbage-First Garbage Collector." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html>

[14] "Java Native Interface Specification." Oracle Java SE Documentation. <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

[15] "Java Native Access (JNA)." <https://github.com/twall/jna#readme>

[16] "BridJ: Let Java & Scala Call C, C++, Objective-C, C#..." <http://code.google.com/p/bridj/>

[17] "SWIG." <http://www.swig.org/>

[18] "HawtJNI." <http://fusesource.com/forge/sites/hawtjni/>

[19] "javacl - OpenCL bindings for Java." <https://code.google.com/p/javacl/>

[20] "OpenCL - The open standard for parallel programming of heterogeneous systems." <http://www.khronos.org/opencl/>

[21] "OpenCL_FFT." Mac Developer Library, <http://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Introduction/Intro.html>

[22] Nazario Irizarry, "Java/C Comparative Benchmarks." <http://jcompbmarks.sourceforge.net>

[23] Nazario Irizarry, "Poor Man's Timing And Profiling." <http://jtimeandprofile.sourceforge.net>

[24] Nazario Irizarry, "C To Java JNI Shim Generator" <http://cshimtojava.sourceforge.net >