Language Selection for Mobile Systems: Java, C, or Both?

Keith S. Vallerio and Niraj K. Jha
Dept. of Electrical Engineering
Princeton University
Princeton, NJ 08544
{vallerio,jha}@ee.princeton.edu

***Abstract*— For many years, C has been known as a fast, yet unfriendly language. Similarly, Java presents its own trade-offs, including more advanced language features at the cost of slower execution. As Java implementations continue to mature, this distinction has become less clear. Special hardware, better libraries and more sophisticated compilers have placed Java-based program execution times in the same realm as C-based programs. This paper demonstrates that superior performance is obtained by carefully selecting the appropriate language for implementing the system. In some cases, this will involve both languages interacting synergistically. For the SciMark 2.0 benchmark suite, using the Java Native Interface (JNI) increases performance by 2.29X for the best case and 1.29X on average compared to the C implementation on a Sharp Zaurus PDA. Simultaneously, the energy improvement for the best case is 2.26X and 1.24X on average.**

I. Introduction

Although processing capabilities of mobile embedded systems have increased dramatically, battery technology has not kept pace, making it important for designers to reduce energy consumption wherever possible. The research presented in this paper focuses on analyzing the effect that language selection (between C and Java [1]) has on embedded system energy consumption. Since its inception in 1995, much work has gone into improving the performance of Java-based software, resulting in a language that is competitive with C. Since their strengths are not symmetric, the optimal system implementation may require that one task be written in C and another in Java. Furthermore, the relative performance of the two languages is based not only on the application, but on the underlying system architecture and available compilers as well. The main contribution of the work contained in this paper is to:

1) compare the relative performance of C and Java, and
2) analyze the efficacy of combining C and Java to enhance performance and reduce energy consumption.

This work demonstrates that embedded system developers can capitalize on the strengths of both C and Java by selecting the implementation language based on the task and system architecture. The methodology presented in this paper differs from previous work since it analyzes and demonstrates the efficacy of using multiple programming languages for embedded system applications. For the SciMark 2.0 benchmark suite [2], invoking just a few key routines in Java from C programs resulted in a 1.29X performance increase on average and 2.29X in the best case. Furthermore, the development time cost for integrating the two languages can be amortized across multiple projects since standard application programming interfaces (APIs) can be used.

The remainder of this paper is organized as follows. Section II presents previous work related to energy reduction for embedded systems and increasing Java code performance. Section III describes compilation methods for C and Java applications as well as the use of the JNI. Section IV presents the experimental setup along with the results. Section V presents an analysis of the results. Section VI contains a synopsis of the work described in this paper.

II. Related Work

Section II-A presents an overview of previous energy reduction techniques for mobile embedded systems. A summary of recent advances in Java is presented in Section II-B.

A. Energy Reduction Techniques

The limited battery lifetimes of most mobile embedded systems have led to the development of several techniques for reducing energy consumption. Since an embedded system consists of more than just a microprocessor, a plethora of hardware components have been targeted for energy reduction. Among the most common components are the CPU, hard drive, display, backlight (for PDAs), wireless card, and memory [3]. Although its energy consumption usually accounts for less than 50% of the total system energy, the primary target for energy optimization has been the CPU. CPU energy can be significantly reduced by using voltage scaling when performance is not critical and sleep modes for long stretches of idle time [4]–[6]. The energy consumption of peripherals can also be reduced through the use of low power and sleep modes [7]. Although these techniques require support from the operating system (OS), efficient OS design can further reduce system energy via power-aware resource usage for communication devices [8]. By optimizing the interaction between the OS and the application, the energy lost due to system calls can also be significantly reduced [9]. These methods are used by system designers and are highly effective at reducing system energy consumption.

In addition to power-saving measures, reducing the total execution time is an effective method for reducing system energy consumption. Faster task completion rates allow the system to progress to the next task more quickly or power down to conserve energy. The simplest method for reducing execution time is to use well-known general compiler transformations in current literature [10], [11]. Since embedded system tasks are often known *a priori*, application-specific transformations may be useful in further increasing performance [12], [13]. These techniques can be applied across programming languages and embedded system architectures.

B. *Recent Advances in Java*

Java achieves portability by storing compiled code in a *class* file, which is a machine-independent format known as bytecode. This mechanism for storing executable code has given Java a reputation for being a slow language, since programmers often view Java as an interpreted language, even though just-in-time (JIT) compilers have been available for a long time. However, several improvements have dramatically improved the Java program performance. In [14], methods were developed to speed up traditional compiler optimizations so they could be included in the JIT compiler. In [15], Java numerical computation performance was enhanced by linking Java code with the native libraries via the JNI provided by Sun's Java development kit. Their method of using the JNI is described in Section III-C.1 and differs from the methods we use, which are described in Sections III-C.2 and III-C.3.

Adding specialized hardware or extensions to the language can also help Java overcome some of its bottlenecks. In [16], a buffer is added to cache previously translated bytecode. The work in [17] presents methods to enhance multidimensional array performance by using a superset of Java. Sun's HotSpot Java Virtual Machine (JVM) [18] reduces compilation time for rarely used code segments and uses dynamic compilation to optimize code based on values known only at run-time. The work contained in this paper differs from previous research since it investigates the performance strengths of C, Java, and various combinations of the two.

III. Compilation and Execution Methodologies

A full discussion of compiler theory is not provided here since this paper addresses system-level design considerations. However, a brief description of relevant C and Java compilation procedures is presented to provide the reader with the necessary background material.

Section III-A presents the design flow for compiling and executing programs written in C. Section III-B presents the design flow for compiling and executing Java programs. Section III-C presents the design flow for using the JNI to compile programs composed of both C and Java.

A. *Compiling C Programs*

The standard compilation process for C programs turns source files into static object code. Although the compiled code can take many forms (i.e., stand-alone executables, static libraries, dynamically linked libraries), it is not modified once the compilation process has completed. As a result, much research has been devoted toward improving the efficiency of the code generated during the compilation stage. Recently, more ambitious methods for improving compilation effectiveness have been developed based on feedback-directed optimization (FDO). FDO is an optimization method that uses the results of previous run-time values to optimize code based on the expected input. Readers interested in FDO techniques are referred to [12]. FDO techniques can be very effective at improving performance when trace data are available, but most modern compilers only support basic FDO techniques and system designers do not always have access to characteristic training data. FDO is mentioned here to highlight that the compiler community recognizes that significant improvements in performance can be achieved via FDO techniques.

B. *Compiling Java Programs*

Programs written in Java can be either compiled directly into binary executables or into machine-independent *class* files. Fig. 1 depicts the two flows for the compilation process. The top flow, using *gcj*, produces a native executable in the same manner as the C build process. The bottom flow, using *javac*, creates a *class* file, which contains the Java bytecode. To execute the code contained in a *class* file, the user invokes the JVM and passes it the *class* file as a parameter. Fig. 2 depicts the interaction between the application code contained in the *class* file and the underlying hardware. The Java application code invokes API procedure calls, which are interpreted by the JVM. The JVM translates these commands into instructions for the target architecture and passes them to the hardware

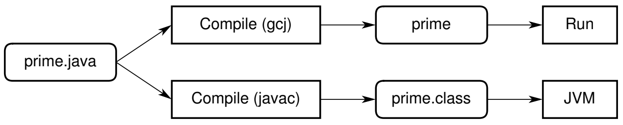

Fig. 1. Java build and execution flow graph

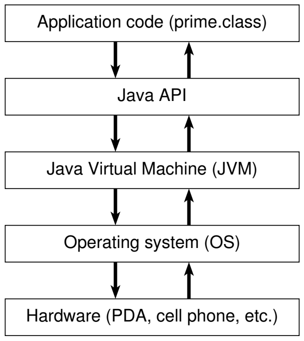
Application code (prime.class)
Java API
Java Virtual Machine (JVM)
Operating system (OS)
Hardware (PDA, cell phone, etc.)

Fig. 2. Running Java program via the JVM

via the OS. The hardware executes the translated instructions and the results return, through the OS, to the JVM, and finally to the application code.

The execution process described in Fig. 2 can be enhanced through the use of dynamic compilation. JVMs that use JIT compilation recompile the bytecode for the target architecture as part of the execution process. The recompilation can be performed at the start of execution or each time uncompiled code is encountered (hence the name just-in-time). Another methodology, used by Sun's HotSpot JVM, is to only compile sections of code that are accessed frequently. These JVMs are able to utilize aggressive FDO techniques since they recompile frequently used code as it is being executed. In summary, using a JVM incurs the overhead of compiling at runtime, but enables more optimization techniques that cannot be used on statically compiled code.

C. Java Native Interface

Using the JNI, Java applications can link with existing libraries and other binaries implemented in another language (i.e., C) through a series of API calls. Invoking procedures from native binaries (written in C) to enhance the performance of Java applications is not new. However, this work proposes using the JNI API in the other direction (C applications invoking Java procedures) to improve performance.

1) Java calling C: This section presents the basics of building and executing applications that combine C and Java via the JNI. Fig. 3 depicts the flow for building application code that uses the JNI. The diagram has two concurrent flows that begin with the C and Java source files. The left side of the diagram presents the flow for compiling

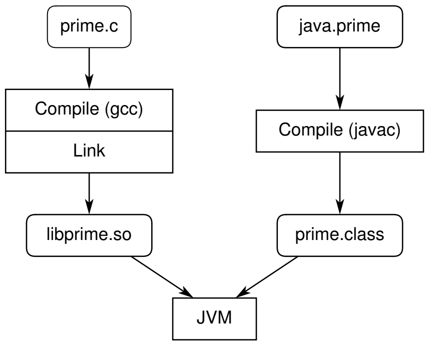

Fig. 3. Building and running a Java application (using the JNI)

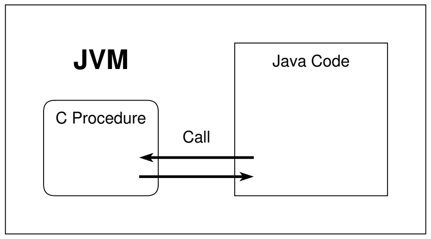

Fig. 4. Java application calling C procedure

the C file, *prime.c*. This procedure is similar to the standard build procedure for C programs, but results in a library (*libprime.so*) instead of an executable. The right side of the diagram presents the standard Java compilation to bytecode flow. To run the application, the user starts the JVM with the *class* file as input.

The traditional interaction between Java code and C code is depicted in Fig. 4. In this case, the JVM is invoked and given a *class* file as a parameter. Before the Java application code can invoke the procedures written in C, the application code must instruct the JVM to load the library containing the compiled C code. Once this is accomplished, the procedures written in C may be invoked by the Java application code.

2) C calling Java: It is also possible to build C programs that invoke Java procedures (contained in a *class* file). Since applications written in C compile into binary executable files that are standalone, the build process must include JVM libraries in addition to the application code contained in the *class* file. Fig. 5 depicts the flow for this compilation process. The Java application code is compiled separately into a *class* file as described previously in Section III-B. The C application code is compiled and linked with the Java libraries.

To run the application, the user runs the C binary file. The C application contains code that initializes the JVM and instructs it to invoke the application's Java procedures found in the *class* file. The relationship between the two languages

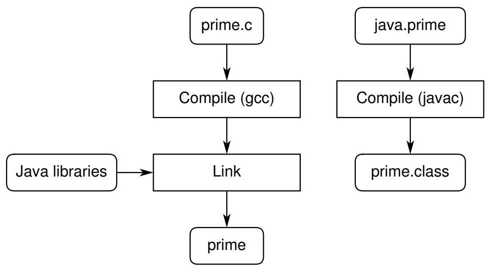

Fig. 5. Building C application (using the JNI)

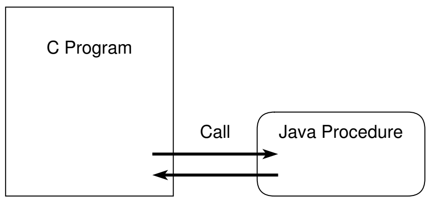

Fig. 6. C application calling Java procedure

is depicted in Fig. 6.

3) Java-wrapped C: The method described above requires library files (containing code for the JVM) for the C application to invoke the JVM. However, these library files may not be available. In this case, compilation proceeds in the same manner as described in Fig. 3. The difference between the traditional method and the one described here is that a Java wrapper is added around the original C application. Fig. 7 depicts the interaction between the two languages. Using this method, the user invokes the JVM and passes it the *class* file containing the Java wrapper code. The wrapper, in turn, invokes the main procedure of the original C application and passes the run-time arguments to that procedure. The C application executes normally and is able to invoke the Java procedures via the JNI API. When the C application terminates, control returns to the Java wrapper code, which then terminates the program.

IV. Experiments

The experimental setup is described in Section IV-A. Section IV-B presents the results for a series of microbenchmark programs. Section IV-C presents the results for *floating-point* application kernels.

A. Experimental setup

1) Hardware: The mobile computing system used for the experiments was a Sharp Zaurus SL-5500 PDA running Embedix, a Linux-based embedded OS. The Zaurus PDA is equipped with a 206MHz StrongARM SA-1110 CPU, 64MB SDRAM and 16MB FLASH ROM [19]. The Zaurus PDA was used for this investigation since it is a modern mobile computing system that is both

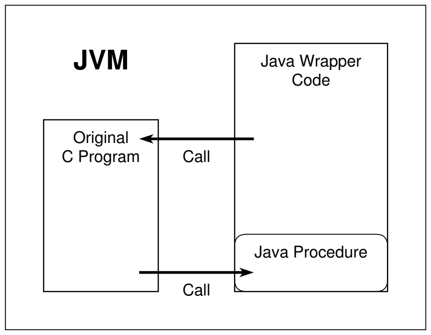

Fig. 7. Running a C program (calling a Java procedure) inside the JVM

compact and powerful. Since it is capable of using a wireless card, the Zaurus can be used standalone or as part of a connected system.

2) Benchmark compilation: The bytecode for the Java implementation was generated via Sun's Java 2 runtime environment, standard edition software development kit (J2SE 1.4.2 SDK) [1] and run using Jeode Virtual Machine. The Jeode Virtual Machine uses an on-the-fly compiler that is designed for mobile computing devices. It is available for a range of processors (ARM, MIPS, x86 and PowerPC) and OSs (Windows CE, Linux, uITRON, VxWorks, and Windows NT) [20], which makes it a good target for investigation.

The JNI implementation used in the experiments was generated using the Java-wrapped C method described in Section III-C.3.

The standard C implementation was compiled using the GNU compiler for the ARM architecture with the following options: -O3 -funroll-loops. The output produced by this compiler is referred to as standard C.

An enhanced C implementation was also compiled with an enhanced version of the compiler described in [21]. Since the Zaurus PDA is based on a StrongARM SA-1110, it does not contain a dedicated floating-point unit (FPU) for non-integer calculations. Instead, floating-point operations (FPE instructions) are compiled by the compiler into system calls. Every floating-point operation requires a context switch for the OS to handle the floating-point computation. It is possible to reduce the overhead by linking the application binary with a software floating-point library. Using this technique generates binaries that are noticeably faster (3-4X). The output produced in this manner is referred to as enhanced C. *Note that this enhanced version is used as the base when reporting speedups relative to C.*

3) Measurement: Execution time measurements are based on the standard Linux *time* utility. *time* reports the user, system, and real time used by a process. Since the differences in the execution times in the examples are large, the accuracy of this utility is more than sufficient. While these results focus on performance, recent work indicates that there is a strong correlation between performance improvement and total system energy reduction. According to JouleTrack, the variation in power is within 8% for their benchmarks [22]. However, since the total power consumption can vary, we also performed experiments to verify that our methodology reduces energy consumption as well. The energy consumption was measured using a hardware data acquisition card attached to a Windows PC using the data acquisition program *Labview*. The setup is identical to the one used in [23].

B. *General experiments*

In order to compare the relative performance of each implementation language, a series of microbenchmark programs were evaluated. The benchmarks used include:

- *baseline*: Empty program (exits immediately)
- *print*: Prints out a string of 1's
- *addi*: Integer addition
- *multi*: Integer multiplication
- *addf*: Floating-point addition
- *multf*: Floating-point multiplication

The first benchmark, *baseline*, measures the time required to execute a program that performs no computation. This value is negligible for the C implementations, but invoking the JVM (for the Java and JNI implementations) incurs a 1.60s overhead on average. The results of evaluating the other benchmarks are depicted in Figs. 8–10. The X-axis for each of these figures indicates the number of vectors used and is plotted on a logarithmic scale. The Y-axis indicates execution time in seconds. The results of evaluating *multi* and *multf* are not presented since these results are similar to the addition benchmark results.

In Figs. 8 and 9, the C, Java and JNI implementations are compared for integer operations[1]. It can be seen that the Zaurus prints output to the screen and performs integer addition (and multiplication) most efficiently in C. However, with the exception of the 1.60s overhead required to invoke the JVM, the JNI implementation and the C implementation have almost identical performance.

In Fig. 10, the standard C, enhanced C, Java and JNI implementations are compared for floating-point addition. It can be seen that the Zaurus performs floating-point addition (and multiplication) much more efficiently in Java. It is also evident that a large portion of the performance increase can be retained by using the JNI. The Java and

[1]There is no enhanced C in these figures since the performance enhancement comes from performing floating-point computations differently (see Section IV-A.2)

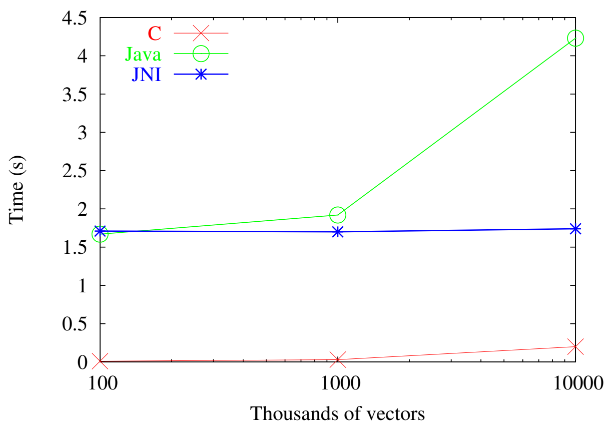

Fig. 8. Printing performance comparison

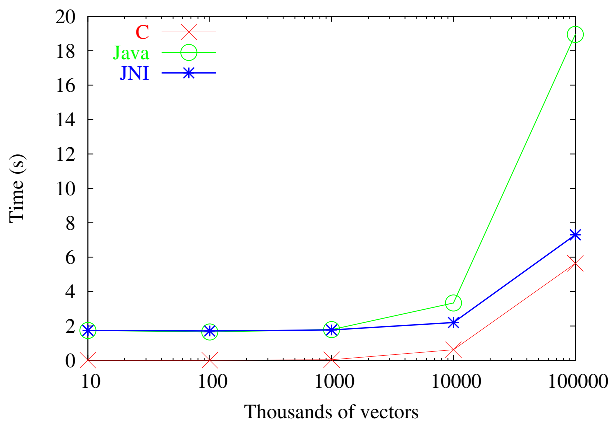

Fig. 9. Integer addition performance comparison

the JNI implementations have similar performance and are over 25% faster than the enhanced C implementation in the best case. For *multf*, the Java and JNI implementations are over 45% faster than the enhanced C implementation in the best case. To further emphasize the difference between the Java/JNI implementation performance and the standard C performance, note that the standard C implementation is so much slower than the others that it goes off the chart. The final value, 610 seconds, is almost an order of magnitude higher than all of the others!

The results presented in Figs. 8-10 indicate that a significant performance gap exists between C and Java implementations. Furthermore, the performance gap is bidirectional, C is faster than Java for integer operations and the reverse is true for floating-point operations. These results can be expanded to include C++ without loss of generality[2].

[2]Several additional experiments were performed to compare C++ and Java implementations. The results of these experiments agree with the data being presented here and have been omitted for brevity and clarity.

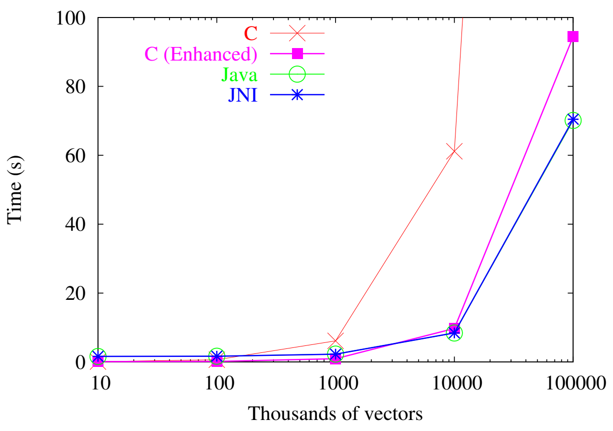

Fig. 10. Floating-point addition performance comparison

C. *SciMark 2.0 experiments*

The results from the microbenchmark experiments were verified by using the SciMark 2.0 benchmark suite. It is comprised of five numerical computation examples that mostly perform floating-point calculations: FFT, Gauss-Seidel relaxation (GSK), Monte Carlo integration (Monte), Sparse matrix-multiply (Smult), and dense LU factorization [2]. SciMark 2.0 was selected because it is a standard benchmark suite and each of the examples in the benchmark suite uses the same algorithm for the C and Java implementations. Thus, it provided means to fairly compare implementation languages. However, note that the SciMark 2.0 examples primarily perform floating-point calculations. Thus, the Java implementations of the SciMark 2.0 examples perform better than the C implementations. The results from Section IV-B predict that this will be true, but Java would not be the best implementation language for integer calculation intensive programs.

The results of the SciMark 2.0 experiments are provided in Table I. Column 1 gives the name of the benchmark. Columns 2 and 3 list the MFLOPS rating for the benchmarks implemented in standard C and enhanced C, respectively. The JNI version is listed in Column 4. Column 5 lists the MFLOPS rating for the Java implementation. MFLOPS measure millions of floating-point operations per second. Thus, higher numbers are better.

It can be seen from the table that the Java implementation has the greatest overall performance. Thus, the ideal implementation language *for these kernels* is Java[3]. However, if these kernels are implemented as part of a system with significant legacy code written in C, using the JNI will improve performance by 5.63X compared to the standard C implementation and 1.29X compared to the enhanced C implementation on average. In the best case, the performance relative to the standard C and enhanced C implementations is 9.45X and 2.29X, respectively. The energy consumption for each example was also measured based on the setup described in Section IV-A.3[4]. On average, using the JNI results in a 82.4% energy saving compared to the standard C implementation and a 22.7% energy saving compared to the enhanced C implementation.

TABLE I
SciMark 2.0 performance results

Exp.	C_S (MFLOPS)	C_E (MFLOPS)	JNI (MFLOPS)	Java (MFLOPS)
FFT	0.10	0.38	0.63	0.79
GSK	0.22	0.91	2.08	2.19
Monte	0.13	0.45	0.20	0.38
Smult	0.18	0.79	1.44	1.56
LU	0.18	0.77	0.18	1.58
Ave.	0.16	0.66	0.91	1.30

Note: Higher values are better.

V. Discussion

The original focus of this project was to determine a new way to leverage C application speed on existing Java code since it was assumed that C implementations of numerical computation are more efficient than their Java counterparts. The results of the integer experiments presented in Section IV are consistent with this assumption. However, as presented in Section IV, Java implementations on the Zaurus excel at floating-point operations. (Experiments on other architectures, omitted for the sake of brevity, verified that this property was a function of the Zaurus not having an FPU.)

The results presented in this paper generate new optimization possibilities. Besides tweaking an algorithm or streamlining code, it is possible to increase performance (ranging from a few percent to several times) by changing the implementation language. At first, this seems impractical since developers are loathe to port large volumes of legacy code from one language to another. However, for many systems it is possible to achieve the speed-up for the computationally-intensive portion of the code by taking advantage of the JNI. The rest of the code can remain unchanged.

[3]As mentioned previously, this is due to the applications being dominated by floating-point calculations. Based on the results in Section IV-B, the ideal implementation language would be different for applications that were dominated by integer calculations.

[4]SciMark 2.0 benchmarks vary the size of the computation to force them to run for at least two seconds. Energy measurements were performed by modifying the benchmarks to perform a fixed number of iterations and measuring the current while the benchmarks were running.

A discussion of the internal workings of compilers, beyond the basics presented in Section III, is not included in this paper since this work focuses on a higher level of the design hierarchy. Furthermore, it is clear that the sophistication of compilers for C and Java will continue to increase. However, the relative performance of the two languages may not remain steady, especially as the underlying hardware continues to evolve. For instance, the Zaurus, based on a StrongARM SA-1110, is different than other architectures (such as Sparc) since it does not include an FPU. As a result, the relative performance of C and Java implementations is a function of the type of application, the system architecture, and the relative sophistication of the compilers. System designers can use microbenchmarks to determine the strengths and weaknesses of their system and compiler. Based on that information, they can use the appropriate implementation language(s) to capitalize on the strengths of the system. Using this methodology provides system-level optimization opportunities that can noticeably increase performance. For embedded systems, where performance and energy are often primary constraints, this gives designers yet another option for optimizing their system.

VI. Conclusion

This work demonstrates that appropriately selecting the implementation language can increase mobile system application performance and often results in a correlating decrease in system energy consumption. For the SciMark 2.0 benchmark suite, it was demonstrated that using a JNI approach can increase floating-point performance of C code by 1.29X on an average and 2.29X in the best case (relative to the enhanced C compiler). Simultaneously, the energy improvement is 1.24X on an average and 2.26X in the best case. The results from the microbenchmarks indicate that a similar improvement can be expected when incorporating C into Java implementations for integer calculation intensive programs.

In the process of generating the examples in Section IV-B, it was found that standard wrappers for JNI routines could be reused for each of the benchmarks in that section. Thus, the integration of the two languages can be streamlined by developing a set of APIs. Developers working with legacy code can use the JNI for computation-intensive kernels, providing them with enhanced performance (ranging from a few percent to several times), without having to port the whole application. The main contribution of this work is that implementation language selection can have significant impact on system performance and energy consumption.

References

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

[2] *http://math.nist.gov/scimark2/*.

[3] J. Lorch, "A complete picture of the energy consumption of a portable computer," Master's thesis, University of California at Berkeley, 1995.

[4] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. Symp. Operating Systems Design and Implementation*, Nov. 1994, pp. 13–23.

[5] M. Srivastava, A. Chandrakasan, and R. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. VLSI Syst.*, vol. 4, no. 1, pp. 42–55, Mar. 1996.

[6] J. Lorch and A. J. Smith, "Scheduling techniques for reducing processor energy use in MacOS," *Wireless Networks*, pp. 311–324, Oct. 1997.

[7] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis, "Power aware page allocation," in *Proc. Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 105–116.

[8] A. Vahdat, A. Lebeck, and C. S. Ellis, "Every joule is precious: The case for revisiting operating system design for energy efficiency," in *Proc. SIGOPS Euro. Wkshp.*, Sept. 2000.

[9] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Analysis of power dissipation in real-time operating systems," *IEEE Trans. Computer-Aided Design*, vol. 22, no. 5, pp. 615–627, May 2003.

[10] J. Bently, *Writing Efficient Programs*. Englewood Cliffs NJ: Prentice-Hall Inc., 1982.

[11] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[12] M. D. Smith, "Overcoming the challenges to feedback-directed optimization," in *Proc. Wkshp. Dynamic and Adaptive Compilation and Optimization*, Jan. 2000, pp. 1–11.

[13] W. Wang, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "Input space adaptive embedded software synthesis," in *Proc. Int. Conf. VLSI Design*, Dec. 2002, pp. 711–718.

[14] M. Cierniak and W. Li, "Just-in-time optimizations for high-performance Java programs," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1063–1073, Nov. 1997.

[15] S. Mintchev and V. Getov, "Automatic binding of native scientific libraries to Java," in *Proc. Int. Symp. on Computing in Object-Oriented Parallel Environments*, Dec. 1997, pp. 129–136.

[16] R. Radhakrishnan, R. Bhargava, and L. K. John, "Improving Java performance using hardware translation," in *Proc. Int. Conf. Supercomputing*, June 2001, pp. 427–439.

[17] C. van Reeuwijk, F. Kuijlman, and H. Sips, "Spar: A set of extensions to Java for scientific computation," *Concurrency and Computation: Practice and Experience*, pp. 277–299, Mar. 2003.

[18] http://java.sun.com/products/hotspot/

[19] *http://www.sharp-usa.com/*.

[20] *http://www.esmertec.com/products/jeode_faq.shtm*.

[21] http://www.ee.princeton.edu/∼wqin/build.htm

[22] A. Sinha and A. Chandrakasan, "Jouletrack - A web based tool for software energy profiling," in *Proc. Design Automation Conf.*, June 2001, pp. 220–225.

[23] L. Zhong and N. K. Jha, "Graphical user interface energy characterization for handheld computers," in *Proc. Int. Conf. Compilers, Architectures & Synthesis for Embedded Systems*, Oct. 2003, pp. 232–242.