# How Programming Languages and Paradigms Affect Performance and Energy in Multithreaded Applications

Guilherme G. Magalhães, Anderson L. Sartor, Arthur F. Lorenzon, Philippe O. A. Navaux, and Antonio Carlos Schneider Beck
Universidade Federal do Rio Grande do Sul (UFRGS) – Instituto de informática, Porto Alegre – Brazil
*{ggmagalhaes, alsartor, aflorenzon, navaux, caco}@inf.ufrgs.br*

*Abstract*— **Considering that multithreaded applications may be implemented using several programming languages and paradigms, in this work we show how they influence performance, energy consumption and energy-delay product (EDP). For that, we evaluate a subset of the NAS Parallel Benchmark, implemented in both procedural (C) and object-oriented programming languages (C++ and Java). We also investigate the overhead of Virtual Machines (VM) and the improvement that the Just-In-Time (JIT) compiler may provide. We show that the procedural language has better scalability than object-oriented ones, i.e., the improvements in performance, EDP, and energy savings are better in C than in C++ and Java as the number of threads increases; and that C can be up to 76 times faster than Java, even with the JIT mechanism enabled. We also demonstrate that the Java JIT effectiveness may vary according to the benchmark (1.16 and 23.97 times in performance and 1.19 to 19.85 times in energy consumption compared to the VM without JIT); and when it reaches good optimization levels, it can be up to 23% faster, consuming 42% less energy, and having an EDP 58% lower than C++.**

*Keywords— **Programming paradigms, performance and energy evaluation, multithreaded applications***

## I. Introduction

While the increasing complexity of applications demands more computing power, energy consumption has also become a primary concern: while most of the embedded devices are mobile, heavily dependent on battery, and operate in constrained environments in which memory, storage, and processing capability are limited (e.g., smartphones, tablets, etc.); general-purpose processors (GPPs) are being pulled back by the limits of thermal design power (TDP). Therefore, the main objective when designing applications is to improve performance with minimal impact on energy consumption [1]. Therefore, several axes must be evaluated when designing a new application.

In order to efficiently exploit the available cores that are present in current processors, multithreaded applications are designed to improve performance by executing parts of the same program concurrently, exchanging data at runtime through shared variables or message passing. For that, several programming paradigms may be used in the development process, in which procedural and object-oriented are the most common. Even though, in several cases, procedural programming provides better performance and more efficient memory usage than object-oriented languages [2][3], the object-oriented methodology is widely applied due to its ability to improve modularity and reusability, which are extremely important for complex applications [4]–[7].

Moreover, applications may be executed on several platforms that are substantially different from each other. While in some languages (e.g. C and C++) code is compiled directly for the target architecture allowing deeper optimization; other programming languages require virtual machines so the code can be executed on several platforms (e.g., Java). This flexibility comes with the cost of an additional layer during execution to transform the code.

In the aforementioned scenario, the application's performance and energy consumption are highly influenced by the way it was developed and by the architecture it targets. Therefore, this work aims at assessing the impact of different programming paradigms (e.g., procedural and object-oriented), Just-in-time (JIT) compilation, and the virtual machine overhead for multithreaded applications implemented in C, C++, and Java. For that, they are evaluated regarding performance, energy consumption, and energy-delay product (EDP) running with 1, 2, 3, 4, and 8 threads.

The benchmarks used in this work comprise a subset of the NAS Parallel Benchmark [8], a well-known suite used for evaluation of parallel computers and applications. They were implemented in C language (procedural) by [9], and the object-oriented versions were developed in Java by [10]. In order to allow the comparison of object-oriented applications that use virtual machines (Java) to the ones that are compiled (C++), we have also implemented these same applications in C++.

In this work, we demonstrate that:

- Procedural language has better scalability than the object-oriented ones, i.e., the improvements in performance, EDP, and energy savings are better in C than in C++ and Java as the number of threads increases. For instance, C is 1.75 times faster and consumes 44% less energy than C++ in the sequential execution; while in the execution with 8 threads, C is 4.9 times faster and consumes 75% less energy than C++.
- Depending on the application behavior, Java running with the JIT compiler is able to outperform C++ in both performance and energy consumption.
- The importance of the JIT compiler by demonstrating that it is able to optimize the performance, energy, and EDP of Java execution in all cases. However, improvement factor

varies from 1.16 to 23.97 times in performance, and from 1.19 to 19.85 times in energy consumption.

The remaining of this work is organized as follows. Related work is discussed in Section II. The programming paradigms used in this work are presented in Section III. The methodology is discussed in Section IV, where the benchmarks and how they were parallelized are presented. In addition, this section also presents the execution environment. Section V discusses the results. Finally, Section VI draws the final considerations.

## II. Related Work

### A. *Single-Threaded Applications*

The most comparisons between C, C++, and Java documented considers only the performance of sequential applications. In [11], the authors quantify the performance difference between Java and C++ in robotic applications. The results show that Java is from 1.09 to 1.91 slower than C++. Although C++ has the best performance, the authors concluded that Java offers a set of interesting features, such as portability, reusability, and maintainability. The execution time of C++ and Java are also compared in [12]. By using four sorting algorithms (bubble, insertion, quick, and heap sort), the results demonstrated that C++ was faster than Java from 1.45 to 2.91 times. In the same way, the authors in [13] show that C++ is about 10% more efficient and faster than Java.

The authors in [14] compare the performance between Java and C running the Linpack benchmark. This benchmark measures how fast a computer solves a dense system of linear equations. The results showed that Java was 2.25 times slower than C in the most significant case. In another comparison between C and Java, the authors in [15] used four applications to measure performance: matrix multiplication, a division-intensive loop, polynomial evaluation, and a distribution function. The results show that C has better performance than Java. A performance comparison between different programming paradigms is performed in [16]. The authors consider C as procedural language, and C++ and Java as object-oriented language. The results showed that, for polynomial multiplications, Java was 21% faster than standard C, and 2.61 times slower than C++.

Considering both performance and power dissipation, the authors in [17] compare object-oriented vs. procedural programming. The work investigates the effect of object-oriented techniques in C++ compared to traditional procedural programming in C-style on embedded systems. The OOPACK Benchmarks, a small suite of kernels that compares the relative performance of C++ and C was used in the experiments. The results show that the execution time and power dissipation of object-oriented programming can significantly be increased when compared to procedural language. The authors in [18] evaluate the Dalvik virtual machine impact in the supported architectures of the Android platform (i.e., ARM, MIPS, and x86). The authors compared Java applications to a version of the application in which the most time consuming methods were implemented in C, therefore, being compiled for the target architecture and not being interpreted by the virtual machine. The results highlight that the virtual machine presents different overhead when executed in different processor architectures.

### B. *Multithreaded Applications*

Few studies investigated the potential of using parallelism exploitation to speedup object-oriented applications. The authors in [19] evaluated the performance of Java thread using the embarrassing parallel (EP) kernel from NAS Parallel Benchmark. The tests were performed on two multithreaded platforms. The results show that the behavior of Java application is determined by the mapping of Java thread to the system native threads. Also, the authors concluded that the use of an excessively large number of threads should be avoided whenever possible, because of significant thread management overheads. Considering different parallel programming interfaces for C language, the authors in [20], [21] and [22] investigate the performance and energy consumption when running parallel applications on different multicore processors and architectures.

Considering the execution of object-oriented applications on massively parallel architectures, the authors in [23] assess the performance and productivity of Java. As a case study, they consider two representative Java GPGPU projects: jCuda [24] based on Cuda; and Aparapi [25], based on OpenCL. To evaluate the performance, the authors used four kernels from the Scalable Heterogeneous Computing Benchmark Suite (SHOC): Matrix multiplication, stencil 2D, and fast fourier transform with single and double floating point precision. The authors concluded that while Aparapi presents a good tradeoff between productivity and performance, jCuda provides better performance results at the price of more programming effort. In

Table I. Comparison of our contributions with the related work

| | | [11]–[13] | [14]–[16] | [17], [18] | [19], [23], [26] | Our work |
|---|---|---|---|---|---|---|
| Environment execution | Object-Oriented | x | x | x | x | x |
| | Procedural | | x | x | | x |
| | JIT and VM evaluation | | | | | x |
| | Single-threaded applications | x | x | x | x | x |
| | Multithreaded applications | | | | x | x |
| Evaluated metrics | Performance | x | x | x | x | x |
| | Energy | | | x | | x |
| | Energy-delay product | | | | | x |

the same way, the authors in [26] compared two parallel versions implemented in java running on the GPU and CPU. The results show that the version running on GPU has a speedup of 6 times against CPU serial implementation, and speedup of 2 times using java parallel implementation.

### C. *Our Contributions*

Table I presents a comparison of the environment and metrics evaluated in this study to the contributions discussed in the related work. Few works investigate the potential using of multithreaded exploitation to speedup object-oriented applications, and they do so in a limited environment and/or restricted number of metrics [19], [23], [26]. Even though the authors in [17] perform an evaluation of performance and energy on both programming paradigms, they evaluate only single-threaded applications. As it can be observed in Table I, this paper extends previous works by evaluating the behavior of different programming paradigms on the performance, energy, and energy-delay product of multithreaded applications; considering native execution, the impact of a virtual machine, and the Just-in-time compiler.

## III. Programming Paradigms

A programming paradigm is defined as a style or a way of programming and used to classify the programming languages. Although there are different paradigms, such as constraint, dataflow, functional, logical, among others, the most widely used are procedural and object-oriented ones.

### A. *Procedural*

This paradigm is as a subset of the imperative paradigm but based on the concept of code modularity, in which a large program is divided in smaller reusable procedures. By decomposing a large problem, one can obtain component problems of manageable size to be dealt with one at a time, and each containing a limited number of details [27]. Another important aspect is scoping, where the lifetime of a variable is as short as possible. The most common language is C, which is mainly oriented towards system programming and is close to the machine [28], being the lowest-level language considered in this work.

Parallelism exploitation occurs through different parallel programming interfaces such as POSIX Threads, Cilk++, and OpenMP (Open-Multi Processing), which is the most widely used. It consists of a set of compiler directives, library functions, and environment variables. Parallelism is exploited by the insertion of directives in the sequential code that inform the compiler how and which parts of the code should be executed in parallel [29].

### B. *Object Oriented*

This paradigm aims to make problems more tangible for the human understanding, in which the reality is modeled so that it is divided into classes. These classes are groups of objects with the same characteristics (attributes) and behavior (methods). The modularity and scoping aspects of procedural languages are important in this paradigm as well, since only the methods of the object's class should be able to modify its data. The classes may inherit some of its attributes or methods from a super-class, which allows code reuse and improves maintainability.

The most common languages in this paradigm are C++ and Java. The former aims to deliver the flexibility and efficiency of C with facilities for program organization (usually referred to as object-oriented programming) [30]. It was projected to be a better C, conserving C as its subset. Java is the highest-level language considered in this study. It was designed to look and feel like C++ but with concerns about complexity and portability since applications running over heterogeneous distributed environments are common nowadays [31].

Despite the fact of Java and C++ being object-oriented languages, they are distinct in several points. For example, the C++ compiler translates the source code into machine code which is hardware dependent and will be executed only by the target CPU and operating system. On the other hand, the Java compiler translates source code into bytecode, which is interpreted by the Java Virtual Machine (JVM). The JVM is the component responsible for the hardware independence of the code [32].

Another difference between Java and C++ is the code optimization, while C++ compiler optimizes code in compile time, Java's compiler optimizes it in execution time. The compiler responsible for optimizations in Java environment is the JIT compiler. During code interpretation, JIT compiles the entire method before it is executed on the system and saves the compiled version on a heap for future reuse. Optimizing code during execution allows better performance since it has information dynamically, however, this comes at the cost of time and energy to perform the interpretation when the machine code is not available to be reused.

## IV. Methodology

### A. *Benchmarks*

The NAS Parallel Benchmarks are a well-known suite of benchmarks originally developed by NASA [33]. The NPB programs are derived from computational fluid dynamics codes. There are several versions of the suite designed to evaluate the performance of parallel computers and parallelization tools, such as MPI [8], High-Performance Fortran [34], Java [10], and C [9]. In this work, four parallel applications from NAS PB were chosen:

*Fourier Transformation* (FT) is a 3-D partial differential equation solution using FFTs. This benchmark performs the essence of many spectral codes. It is a rigorous test of communication performance.

*Integer Sort* (IS) is a benchmark that sort N keys using a linear-time integer sorting algorithm based on computation of the key histogram. It tests both integer computation speed and communication performance.

*Conjugate Gradient* (CG) uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive sparse matrix with a random pattern of nonzero. This benchmark tests unstructured computations and communications by manipulating a diagonally dominant matrix with randomly generated locations of entries.

*Low-Upper* (LU) Gauss-Seidel solver is a technique for solving the *n* equations of the linear system of equations $Ax = b$ one at a time in sequence, and uses previously computed results as soon as they are available. LU uses the symmetric successive over-relaxation method to solve the discrete Navier-Stokes equations by splitting it into block Lower and Upper triangular systems.

The Java NAS applications implemented by [10] use Java Threads by inheritance. Every application has one base class, one main class – which performs the whole task in serial version and, in parallel version, creates and synchronizes threads –, and a variable number of working classes, which performs the computation in the parallel version. The synchronization between the threads is guaranteed by the use of wait/notify, and mutual exclusion (synchronized blocks and methods).

The C++ applications were implemented using the C++11 Threads library, which is based on C's library POSIX Threads, following object-oriented guidelines [30]. Like Java, they have one base class, one main class and one or more working classes. To synchronize, mutual exclusion and condition variables were used; and to ensure data integrity on parallel regions, atomic variables were used.

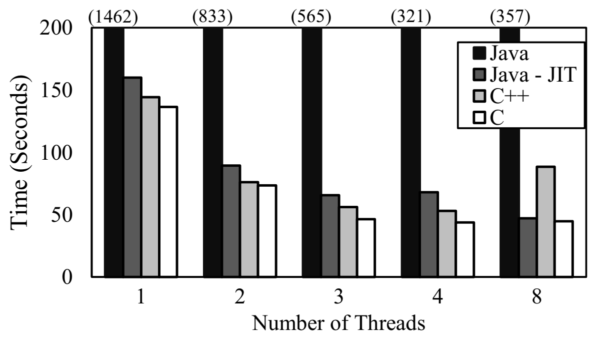
TABLE II. MAIN CHARACTERISTICS OF THE PROCESSOR

| | **Intel Core i7** |
|---|---|
| **Microarchitecture** | Skylake |
| **Frequency** | 3.4 GHz |
| **# Cores** | 4 |
| **# Threads** | 8 |
| **L1 Data Cache** | 4 x 32 KB |
| **L1 Instr. Cache** | 4 x 32 KB |
| **L2 Cache** | 4 x 256 KB |
| **L3 Cache** | 8 MB |
| **RAM Memory** | 32GB |

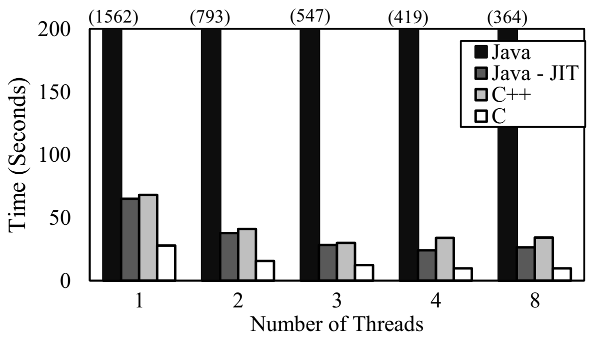
## B. *Execution Environment*

The experiments were performed on the processor Intel Core i7 as shown in Table II. The energy consumption was obtained through the Intel Running Average Power Limit (RAPL) library. RAPL provides a set of hardware counters providing energy and power consumption of the CPU-level components. We are using the following: *package*, which provides the energy of the whole CPU package (core, cache, bus, etc.); and the memory controller *DRAM* [35].

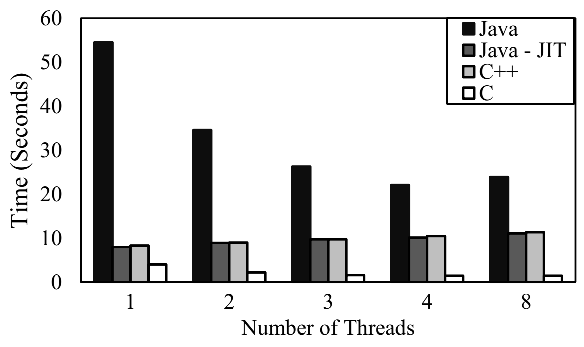
The results presented in the next section consider the average of ten executions, with a standard deviation lower than 1% for each benchmark. The programs were split into 2, 3, 4,



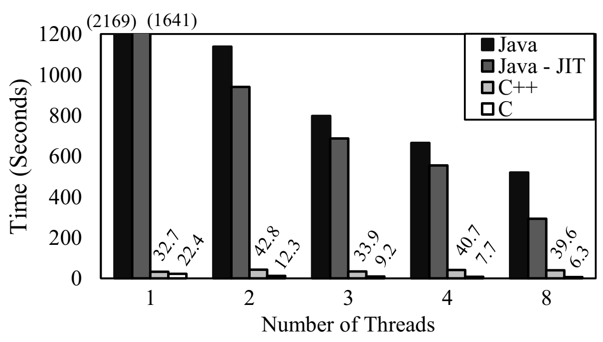
a) Conjugate Gradient



b) Fourier Transformation



c) Integer Sort



d) LU – Lower Upper

Fig. 1. Performance results

and 8 threads, and the benchmarks were executed with the following input sizes (they are represented by classes in NAS the Parallel Benchmark): CG, class C; FT, class B; IS, class C; and LU with the class A.

The Operating System where the tests were performed was the Ubuntu 16.04 kernel 4.4.0-21. The C and C++ compiler used was the GNU GCC-4.9 with the optimization flag -O3. The OpenMP version used in the C applications was the 4.0, since it is fully supported by the GCC version in use. The Java was executed using the Oracle 1.8.0_10. To measure the influence of the JIT in the Java versions (which is enabled by default), we disabled it by using the flag "*-Xint*".

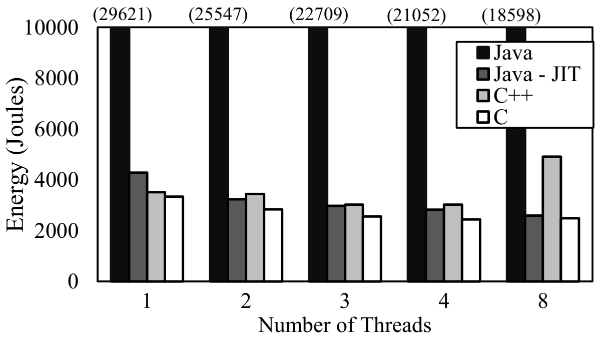
## V. Results

### *A. Performance and Energy Evaluation*

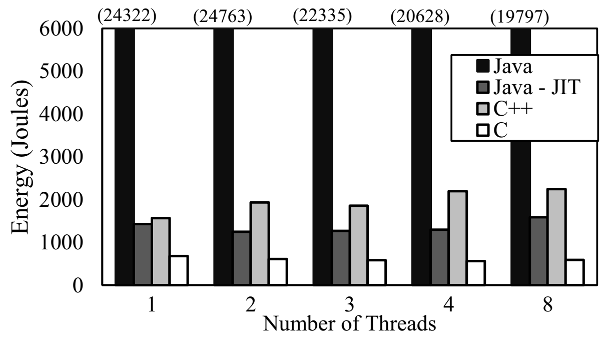
Fig. 1 and Fig. 2 show the execution time (in seconds) and energy consumption (in Joules) when varying the number of threads for the Java (without JIT), Java-JIT (with JIT), C++, and C applications. When comparing the procedural paradigm (C) to the object-oriented (Java and C++), it can be noticed that the C applications were faster and consumed less energy, for any number of threads and benchmark. In the most significant case of performance, C is 97 and 76 times faster than Java and Java-JIT, respectively (LU benchmark - Fig. 1d); and 7.8x faster than C++ (IS benchmark – Fig. 1c). For the most significant case in energy, Java and Java-JIT consumed 79 and 58 times more energy than C, respectively (LU benchmark – Fig. 2d); and C++ consumed 7.9x more energy than C (IS benchmark – Fig. 2c). In addition, C applications present a better scalability than the object-oriented ones, i.e. the improvements in performance, EDP, and energy savings are better in C than in C++ and Java as the number of threads increases. For instance, C is 1.75 times faster and consumes 44% less energy than C++ in the sequential execution; while in the execution with 8 threads, C is 4.9 times faster and consumes 75% less energy than C++. Therefore, applications with lower abstraction levels outperform object-oriented ones in terms of performance and energy consumption in multicore environments.

Let us compare the Java and Java-JIT executions, in which the JIT improved performance and reduced energy consumption in all benchmarks. Table III presents the JIT improvements regarding the performance and energy consumption. The improvement varies from 1.16 to 23.97 times on performance and from 1.19 to 19.85 times on energy. The only benchmark that the JIT is not able to efficiently optimize is the LU, which is executed in a pipelined way and consequently reduces the opportunity of executing already translated code from the JIT heap.

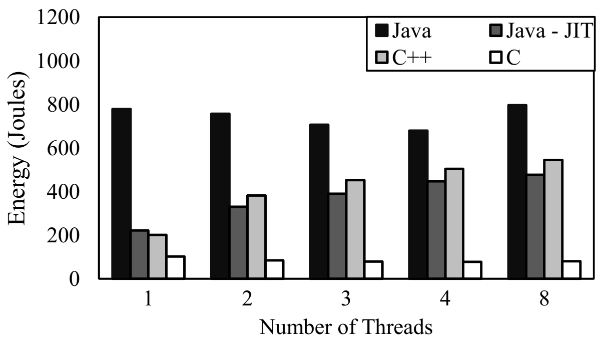
When comparing Java-JIT applications to C++, the C++ had better performance for CG and LU (in which the JIT is not



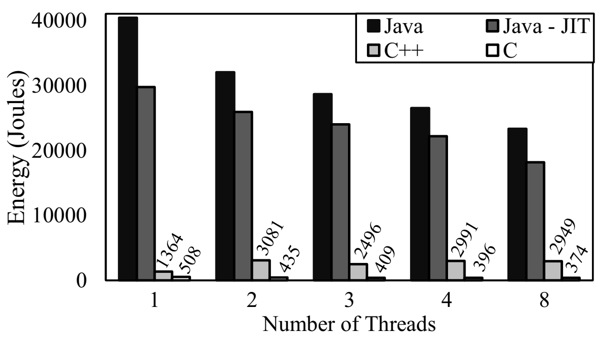
a) Conjugate Gradient



b) Fourier Transformation



c) Integer Sort



d) LU – Lower Upper

Fig. 2. Energy results

efficiently exploited), and similar performance on FT and IS. When the energy consumption is considered, C++ consumed 44% more energy than Java-JIT for the FT benchmark on average due to synchronization issues. On the other hand, for LU, C++ consumed 88.7% less energy on average than the Java-JIT due to the inefficiency of the JIT exploitation. Therefore, for some benchmarks, the improvements that JIT provides allow the Java code to have competitive performance and energy consumption when compared to C++ applications, with the former having the advantage of being multiplatform without the need for recompilation.

The procedural paradigm has better scalability in terms of performance and energy consumption when the number of threads increases, and the JIT compilation is able to optimize all Java applications. However, the benefits from applying the JIT mechanism greatly depends on the application behavior, not being able to efficiently optimize the execution if the application does not have a cyclic behavior that would use already saved translations of the bytecode. In such cases, Java is up to 50 times slower and consumes 21 times more energy than C++.

## *B. Energy-Delay Product Evaluation*

As shown in the previous section, C language has the best results for both performance and energy. However, when comparing Java and C++, the language that offers the best performance is not necessarily the same that spends less energy. In this case, the EDP may be useful since it correlates both metrics into a unique value, as shown by (1).

$$EDP = Energy\ x\ Delay \tag{1}$$

This metric is being widely used to evaluate different environments such as in [36], [37], since it allows in a unique value, analyze the relationship between energy and performance. For example, let us consider two scenarios: (*i*) an application spends 10 Joules of energy and takes 100 seconds to execute; (*ii*) an application executes in 50 seconds, but spends 40 Joules of energy. The scenario (*i*) has EDP of 1000 while the second one has EDP of 2000. Thus, although the first scenario has been two times slower than the second one, it has better EDP.

Fig. 3 presents the EDP in logarithmic scale for each programming language. Java applications without JIT have larger EDP due to the increased execution time and energy consumption. In addition, the C applications have the lowest EDP because of its reduced execution time and energy consumption. Comparing the Java-JIT to the C++ version, the Java has lower EDP for the multithreaded version of the FT and IS algorithms, and higher EDP for CG and LU, the latter being due to the inefficient application of the JIT mechanism. Hence, for FT and IS applications, Java-JIT is able to provide a better EDP (similar performance with lower energy consumption). Even though the energy consumption for the Java-JIT CG

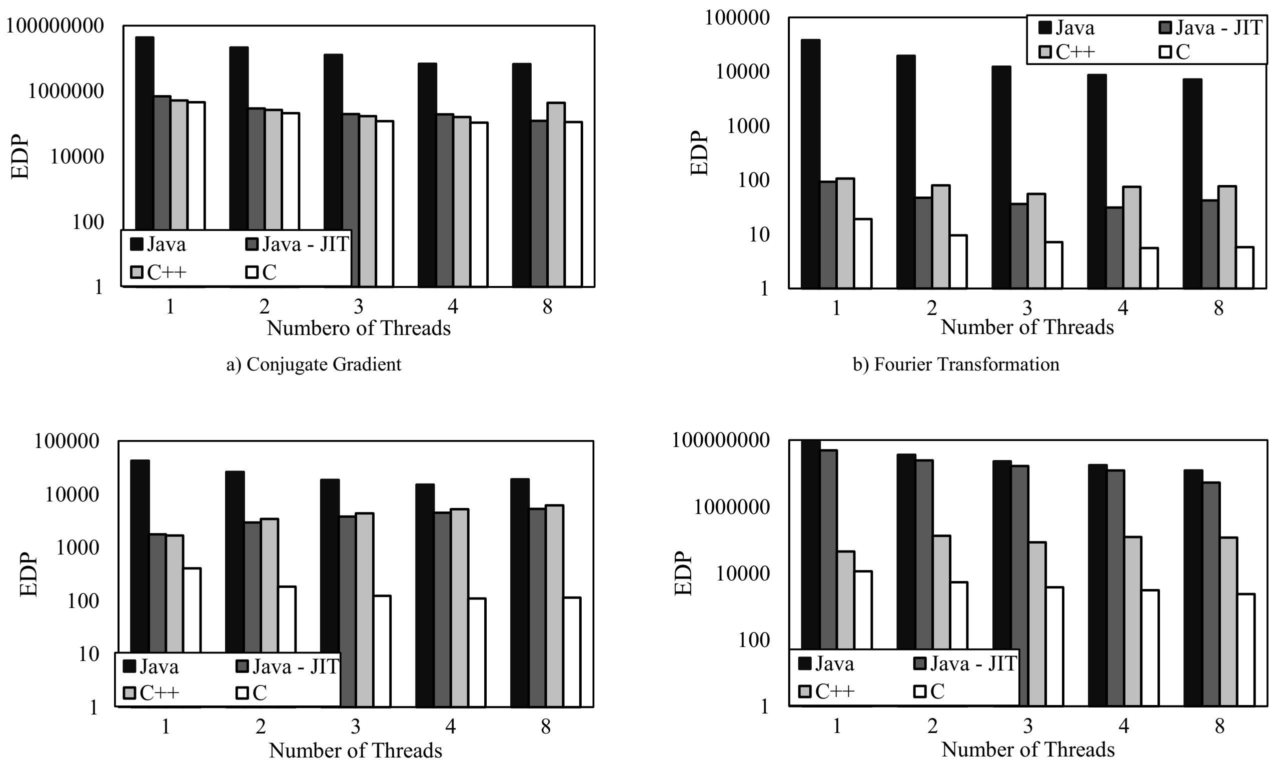


Fig. 3. Energy-delay product results – in logarithmic scale

TABLE III. JIT IMPROVEMENTS ON JAVA VERSIONS - RATIO BETWEEN JAVA WITHOUT JIT AND WITH JIT

| **#T** | **CG** | | | **FT** | | | **IS** | | | **LU** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Time* | *Energy* | *EDP* | *Time* | *Energy* | *EDP* | *Time* | *Energy* | *EDP* | *Time* | *Energy* | *EDP* |
| **1** | 9.14 | 6.92 | 63.23 | 23.97 | 17.02 | 408.21 | 6.87 | 3.53 | 24.23 | 1.33 | 1.35 | 1.81 |
| **2** | 9.33 | 7.39 | 73.71 | 21.03 | 19.85 | 417.37 | 3.87 | 2.29 | 8.86 | 1.21 | 1.23 | 1.49 |
| **3** | 8.62 | 7.63 | 65.72 | 19.28 | 17.57 | 338.82 | 2.71 | 1.81 | 4.91 | 1.16 | 1.19 | 1.38 |
| **4** | 4.73 | 7.44 | 35.22 | 17.37 | 15.93 | 276.87 | 2.19 | 1.52 | 3.34 | 1.19 | 1.19 | 1.43 |
| **5** | 7.60 | 7.18 | 54.59 | 13.77 | 12.49 | 172.01 | 2.17 | 1.67 | 3.62 | 1.77 | 1.28 | 2.27 |

benchmark is lower than the C++ one, the C++ version has better performance, and when both axes are considered, the EDP for the C++ version is better than the Java-JIT for this application.

## VI. CONCLUSIONS

This work analyzed the impact of different programming paradigms (procedural and object-oriented), Just-in-time (JIT) compilation, and the virtual machine overhead for multithreaded applications implemented in C, C++, and Java, taking into account different metrics: performance, energy consumption and EDP. It confirmed that the procedural language C has better results and better scalability than object-oriented ones, i.e., the improvements in performance, EDP, and energy savings are better in C than in C++ and Java as the number of threads increases. On the other hand, the Java version without JIT compiler has the worst results.

In most cases, the Java with JIT and C++ presented similar results. However, in applications that are executed in a pipelined way, which reduces the opportunity of executing already translated code from the JIT heap, Java was significantly slower and more energy consuming than C++. On the other hand, when the application has a great number of synchronization points and data exchange among threads, the overhead to manage these operations was more significant in C++. In these cases, Java with JIT presented better EDP.

As future work, we will expand our benchmark set to cover a larger range of application behaviors and consider other factors, such as Instruction-Level parallelism and applications that are more control or data-flow oriented. We will also compare the impact of using different compilers (e.g., GCC and LLVM – Low Level Virtual Machine) with different levels of optimization. In addition, we will expand the tests so that we can analyze platforms with more TLP exploitation available.

## REFERENCES

[1] A. C. S. Beck, C. A. L. Lisba, and L. Carro, *Adaptable Embedded Systems*. Springer Publishing Company, Incorporated, 2012.

[2] I. Sommerville, *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[3] R. Harrison, L. G. Smaraweera, M. R. Dobie, and P. H. Lewis, "Comparing programming paradigms: an evaluation of functional and object-oriented programs," *Softw. Eng. J.*, vol. 11, no. 4, pp. 247–254, Jul. 1996.

[4] A. D. Robison, "The abstraction penalty for small objects in C++," *Parallel Object-Oriented Methods Appl. 96*, 1996.

[5] A. D. Robison, "C++ Gets Faster for Scientific Computing," *Comput. Phys.*, vol. 10, no. 5, pp. 458–462, Sep. 1996.

[6] B. Calder, D. Grunwald, and B. Zorn, "Quantifying behavioral differences between C and C++ programs," *J. Program. Lang.*, vol. 2, no. 4, pp. 313–351, 1994.

[7] S. W. Haney, "Is C++ Fast Enough for Scientific Computing?," *Comput. Phys.*, vol. 8, no. 6, pp. 690–694, Nov. 1994.

[8] W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "New Implementations and Results for the NAS Parallel Benchmarks 2," in *In 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[9] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012, pp. 341–352.

[10] M. A. Frumkin, M. Schultz, H. Jin, and J. Yan, "Performance and scalability of the NAS parallel benchmarks in Java," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 6 pp.-pp.

[11] L. Gherardi, D. Brugali, and D. Comotti, "A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark," in *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 161–172.

[12] S. Wentworth and D. D. Langan, "Performance Evaluation: Java vs {C}++." 2000.

[13] A. M. Alnaser, O. AlHeyasat, A. A.-K. Abu-Ein, H. (Moh'd S. Hatamleh, and A. A. M. Sharadqeh, "Time Comparing between Java and {C}++ Software," vol. 5, no. 8, pp. 630–633, 2012.

[14] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A Methodology for Benchmarking Java Grande Applications," in *Proceedings of the ACM 1999 Conference on Java Grande*, 1999, pp. 81–88.

[15] P. Sestoft, "Numeric performance in {C}, {C} # and Java." 2014.

[16] L. Bernardin, B. Char, and E. Kaltofen, "Symbolic Computation in Java: An Appraisement," in *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, 1999, pp. 237–244.

[17] A. Chatzigeorgiou and G. Stephanides, "Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors," in *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, 2002, pp. 65–75.

[18] A. L. Sartor, A. F. Lorenzon, and A. C. S. Beck, "The Impact of Virtual Machines on Embedded Systems," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2015, vol. 2, pp. 626–631.

[19] Y. Gu, B. S. Lee, and W. Cai, "Evaluation of {Java} thread performance on two different multithreaded kernels," *Oper. Syst. Rev.*, vol. 33, no. 1, pp. 34–46, Jan. 1999.

[20] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy," *J. Parallel Distrib. Comput.*, vol. 95, pp. 107–123, 2016.

[21] A. F. Lorenzon, M. C. Cera, and A. C. Schneider Beck, "Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems," *J. Signal Process. Syst.*, pp. 295–307, 2014.

[22] A. F. Lorenzon, A. L. Sartor, M. C. Cera, and A. C. S. Beck, “Optimized Use of Parallel Programming Interfaces in Multithreaded Embedded Architectures,” in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 410–415.

[23] J. Docampo, S. Ramos, G. L. Taboada, R. R. Exposito, J. Tourino, and R. Doallo, “Evaluation of Java for general purpose GPU computing,” in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, 2013, pp. 1398–1404.

[24] Y. Yan, M. Grossman, and V. Sarkar, “JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA,” in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, 2009, pp. 887–899.

[25] K. Uchiyama, F. Arakawa, H. Kasahara, T. Nojiri, H. Noda, Y. Tawara, A. Idehara, K. Iwata, and H. Shikano, “Heterogeneous Multicore Processor Technologies for Embedded Systems,” 2012, pp. 11–19.

[26] K. G. Gupta, N. Agrawal, and S. K. Maity, “Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm,” in *Parallel Computing Technologies (PARCOMPTECH), 2013 National Conference on*, 2013, pp. 1–5.

[27] K. Nygaard and O.-J. Dahl, “History of Programming Languages I,” R. L. Wexelblat, Ed. New York, NY, USA: ACM, 1981, pp. 439–480.

[28] D. M. Richie, “The Development of the {C} Language,” unpublished.

[29] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10. MIT press, 2008.

[30] B. Stroustrup, *The {C}++ Programming Language:*, Special. Addison Wesley, 2000.

[31] J. Gosling and H. McGilton, *The {Java} Language Environment*. Sun Microsystems Computer Company, 1995.

[32] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[33] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, “The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 158–165.

[34] M. Frumkin, H. Jin, and J. Yan, “Implementation of NAS Parallel Benchmarks in High Performance Fortran.” 1998.

[35] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, “Power-management architecture of the intel microarchitecture code-named sandy bridge,” *IEEE Micro*, no. 2, pp. 20–27, 2012.

[36] E. Blem, J. Menon, and K. Sankaralingam, “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, no. Hpca, pp. 1–12, 2013.

[37] A. Tiwari, K. Keipert, A. Jundt, J. Peraza, S. S. Leang, M. Laurenzano, M. S. Gordon, and L. Carrington, “Performance and energy efficiency analysis of 64-bit ARM using GAMESS,” in *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing - Co-HPC '15*, 2015, pp. 1–10.