Computer Science and Information Systems 16(2):541–564 https://doi.org/10.2298/CSIS180608009L

Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings*

Mathias Longo[1], Ana Rodriguez[2], Cristian Mateos[2], and Alejandro Zunino[2]

1 University of Southern California,
1337 1/2 W Adams Blvd, Los Angeles (90007), United States
mathiasl@usc.edu
2 ISISTAN-CONICET-UNICEN,
Campus Universitario, Tandil (B7001BBO), Argentina
{ana.rodriguez,cristian.mateos,alejandro.zunino}@isistan.unicen.edu.ar

Abstract. In-silico research has grown considerably. Today's scientific code involves long-running computer simulations and hence powerful computing infrastructures are needed. Traditionally, research in high-performance computing has focused on executing code as fast as possible, while energy has been recently recognized as another goal to consider. Yet, energy-driven research has mostly focused on the hardware and middleware layers, but few efforts target the application level, where many energy-aware optimizations are possible. We revisit a catalog of Java primitives commonly used in OO scientific programming, or micro-benchmarks, to identify energy-friendly versions of the same primitive. We then apply the micro-benchmarks to classical scientific application kernels and machine learning algorithms for both single-thread and multi-thread implementations on a server. Energy usage reductions at the micro-benchmark level are substantial, while for applications obtained reductions range from 3.90% to 99.18%.

Keywords: Energy, Scientific application, Java, Micro-benchmarks, Code refactoring.

1. Introduction

Scientific computing is a field that applies Computer Science to solve scientific problems from other disciplines, such as Mathematics, Engineering, Biology, Physics and Chemistry. Scientific computing is inherently associated with large-scale computer modeling and simulation since it mainly concerns wisely using many computing resources to quickly deliver results for ever-growing problem sizes. In fact, the high popularity of this in-silico approach to research has significantly grown over the last years, which gave birth to Computational Science, a relatively new multidisciplinary

* This is an extended version of https://doi.org/10.1007/978-3-319-31232-3_69

field that uses advanced computing capabilities and notably High-Performance Computing (HPC) infrastructures to solve complex problems.

Irrespective of the computing infrastructure, research in HPC has traditionally focused on executing computations as fast as possible. Much research spanning the high-level architecture of such infrastructures including advances at the hardware level (e.g., more/faster cores for CPUs), platform level (e.g., efficient/robust middleware-level schedulers) and application level (e.g., parallel programming models) has been conducted. Nevertheless, the area has already acknowledged the importance of energy usage as well [6]. Energy consumption accounts for 15% of the operational expenditures in datacenters [17]. Furthermore, the energy consumed in datacenters in Western Europe will increase 100 TWh per year by 2020 [14], which is significant considering that for example 108 TWh was the energy consumption of Netherlands itself during 2014 according to the CIA World Factbook. This leads to huge operational costs, reduced system stability and negative ecological consequences [7].

In response, there is a wide spectrum of research efforts at the hardware level. This involves equipping processors with finer "C-states"/"P-states" and better voltage/frequency scaling techniques. Other ambitious efforts have produced the first ARM-based HPC cluster [35]. Moreover, efforts at the platform level include re-designing operating systems for energy efficiency and providing parallel middlewares to properly trade-off obtained performance and used energy for computations [1]. However, literature shows that there are few efforts focused on how HPC applications should be coded to use less energy [31, 27].

We study the energy consumed by versions of micro-benchmarks representing common programming operations found in scientific applications. To this end, we revisit a recent study [36] that has catalogued such operations but measured their implications in the context of Android programming. The experiments performed in this paper using fixed hardware show that, for the same operation, there are versions which are much more energy-efficient than others. We considered several scientific applications [12] and refactored their implementation code using the energy-efficient versions of micro-benchmarks, again obtaining energy savings. We limit the scope of our research to Java, which is useful for developing HPC applications and middlewares [41] because of its "write once, run anywhere" philosophy. This work is based on an earlier conference version published in [25], but it introduces several pertinent enhancements, namely:

1 A deeper analysis of the reasons behind the obtained differences in energy consumption for the various micro-benchmarks and their variants.
2 The use of representative scientific application kernels (SFA) as scientific test applications by basing on the well-known Phil Colella's categorization [12, 4], who identifies and delineates a set of scientific kernels which form the basis for most of the existing scientific applications. We also consider Machine Learning algorithms, the base of many real-world applications.
3 An active power versus computation time analysis of the above SFAs by considering single-core and multi-core versions of the applications.
4 Statistical significance tests to ensure results validity.

Next Section discusses related works. Section 3 explains the micro-benchmarks and details the SFAs used. Section 4 presents the experimental results. Section 5 presents the conclusions and future works.

2. Related Work

In this section, we will describe relevant efforts to increase energy efficiency in datacenters paying special attention to those focused on the latter, since our goal is reducing energy consumption via code refactorings in HPC applications.

To analyze energy consumption it is necessary to know which hardware resources consume more energy. The main part of power consumed by a server is accounted for the CPU, followed by the memory [26]. Based on this, Chen and Shi [10] present a process-level power profiling tool and a power-aware system module that eliminates energy wasted by abnormal-behavior applications for which hardware information is essential. The authors encourage the design of simple energy models to obtain real and instant measurements to control energy consumed by applications.

Other scientists analyze energy consumption of both hardware manufacture and use, and software execution [2]. For the use phase, Ardito and Morisio [2] present generic guidelines to achieve energy efficiency at four different levels: Infrastructure, Application, Operating System and Hardware. At the application level the guidelines include Design efficient UI, Use event-based programming when possible, Use low-level programming, Reduce data redundancy, Reduce QoS/scale dynamically and Use power/energy profiling tools.

Pinto, Soares-Neto and Castor [33] review works in the area of mobile programming, and they conclude that such works are focused on 6 issues to reduce energy consumption: user interface, CPU offloading, HTTP requests, software piracy, continuously running apps, and I/O operations. The authors also review efforts in the area of parallel programming, identifying 3 issues: excessive copy chains, embrace parallelism and GPU programming. However, authors do not analyze works based on servers. Besides, unlike [2] and [33], we study *concrete* energy-aware programming primitives in HPC code.

The work reported in [38] studies OO design patterns energy consumption in server applications. A new tool for measuring the power consumption and mapping between energy usage and design patterns is proposed. The authors focus on 15 creational, structural and behavioral patterns. Notable conclusions are the usage of design patterns can both increase and decrease the amount of energy used by an application and the usage of design patterns within a category impact energy usage differently.

With regards to application detailed design, Dhaka and Singh [13] study how much the correction of a wrong design affects energy consumption based on code smells, namely god class, feature envy and long method. The authors show that code smell removal permutations yield varying levels of energy consumption for the resulted software versions. It is also observed that the order in which smells are removed affects energy consumption differently. In addition, the authors propose the best sequence that generates a better design code and consumes the least energy possible.

In these lines, some works measure, control and compare energy consumption of languages, libraries, algorithms and applications. The work in [29] presents the

POWERAPI architecture which working together with power modules allows developers to calculate the power consumption of both processes and applications. With this, authors conclude that Java using the default options is quite energy-efficient in comparison to other programming languages, the energy efficiency of Pascal is at the same level as C or C++, and Perl is the most energy-consuming language. The work in [45] goes even further and analyzes execution time, memory consumption and energy consumption of 27 different programming languages over 10 different problems from the Computer Language Benchmarks Game[1]. To increase significance, the authors employ state-of-the-art compilers, virtual machines, interpreters and libraries. The main finding is that C remains as the fastest and most energy efficient language, together with compiled languages in general. In addition, Java is among the top-five most energy-efficient languages, while the least efficient ones are all interpreted.Other works evaluate common practices use or choices when developing applications. Procaccianti, Fernández and Lago [34] evaluate two practices: use of efficient queries (i.e. avoiding indexation mechanisms or unnecessary ordering operations such as SQL 'ORDER BY') and put applications to sleep to reduce CPU (and energy) utilization at the expense of increased execution time. They measure the impact using the Apache WebServer and the MySQL Server. In [27] an exhaustive evaluation of the energy consumption and performance of the NAS parallel benchmarks (NPB) is reported. The authors focus on the impact of multithreading and consider different number of threads and compilers. Authors conclude that it is difficult to balance performance and energy even for relatively simple benchmark as NBP.

Other works study the role of data structures and collections. Energy consumption of operations done on Java List, Map, and Set abstractions (e.g., insertion, iteration, random access) has been evaluated in [19]. Authors found that choosing the wrong Collections type in an application can consume 300% more energy than the most efficient collection. Second, Manotas, Pollock and Clause [24] describe an automated energy optimizer based on code-level changes. Consequently, the authors propose a framework that a) generates different versions of the same code combining all Collections instantiations, b) performs power-monitored executions of all generated versions, c) analyzes the results, and d) generates an optimized version of the original code. In the same line, jStanley [43] is a static code analyzer, implemented as a plug-in for the Eclipse IDE, which focus on reducing energy consumption by replacing Java collections for alternative, more efficient ones. The plug-in finds and quantifies method calls to collections in an application's code (maps, lists, and sets), computes normalized method calls costs, and suggests optimizations. Normalized costs are taken from a previous study from the same authors [44], where they tested the energy costs of 24 implementations of Java sets, lists and maps, considering 42 different methods in total. Interestingly, jStanley allows the user to focus on energy-driven or time-driven optimizations. Reported energy gains using real applications range from 2% to 17%.

3. Common Operations in Scientific Applications

We study eight groups of micro-benchmarks because of their recurrent use in standard and specifically scientific OO programming, namely array copying, matrix traversal,

[1] http://benchmarksgame.alioth.debian.org/

string handling, use of arithmetic operations, exception handling, object field access, object creation and use of primitive data types.

Over the years several built-in facilities were developed in diverse OO languages such as Java and C++ [32]. Then, we determine particularly energy improvement using such facilities to copy an array over implementing manually the same functionality. Additionally, matrices and related operations are important in linear algebra algorithms [28]. Regarding string manipulation, concatenation is the most important operation [11]. Concerning the fourth group, several studies have focused on optimizing arithmetic operations or involve large numbers of them [36]. Exceptions represent a widely used mechanism for elegant error handling. Method invocation was chosen since in OO programming methods must be called to use any subroutine associated with a class. In addition, we chose object creation because it involves costly memory management chores, such as garbage collection in Java or explicit object disposal in C++. Finally, the last group is the use of primitive data types versus (heavier) object-based data types.

3.1 Array Copying (AC)

Most languages include reusable libraries and built-in functionality such as data structure sorting or image manipulation. Using this support has advantages over using ad-hoc implementations since efficiency of such libraries tends to improve over time, which motivated us to compare the use of System.arraycopy method with a manual solution for the same functionality. Arrays are very important in scientific code, e.g. in mathematics arrays are used for representing polynomials.

3.2 Matrix Traversal (MT)

Matrices have many different uses such as writing problems conveniently and compactly or helping to solve problems with linear and differential equations. Additionally, in graph theory an adjacency matrix can be naturally associated to each graph where the position [i,j] indicates if vertex i is connected with vertex j.

Indeed scientific programmers use these structures quite frequently. Matrices are used to store any data type for information handling (i.e., primitive data types or objects) and are a common structure in rendering applications, where they are often used to represent and apply transformations to images. Basically, we tested micro-benchmarks where NxM matrices are traversed by rows and columns. Specifically, both micro-benchmarks involve instantiating a matrix in main memory with numeric values, using a nested loop to iterate the matrix, and accessing each cell while placing the cell value in a local variable.

Java represents n-dimensional arrays by using nested 1-dimensional arrays, which involves in principle more instantiated objects. In addition, the way this nested structure is traversed in a code might exercise the memory hierarchy differently.

3.3 String Handling (SH)

Java applications use the String class to save/read data or display messages to the user. Concatenating smaller data chunks is necessary to create bigger data chunks, thus we work with the "+" operator versus using the StringBuilder class, which exploits buffering. Despite the string concatenation operator is optimized by the compiler using the StringBuilder class, to operate using the String class and its operator "+" might yet be an inefficient practice since each concatenation with this operator implies creating a StringBuilder instance. The operator applied on n strings has $O(n^2)$ complexity, and requires memory space to maintain intermediate concatenations. We consequently expect an energy improvement using StringBuilder.

3.4 Use of Arithmetic Operations (AO)

Arithmetic operations are commonplace in scientific applications. This is illustrated for instance by data compression and mathematical applications. Also, scientific applications often need millions of calculations. Thus, the more energy-efficient the arithmetic operations are, the lower the energy consumption becomes. Since addition is one of the commonest arithmetic operation CPUs solve, we measure energy consumption of adding primitive types (int, long, float and double). Specifically, the micro-benchmark performs the successive addition into a local variable of the content of another variable whose value does not change and is set upon executing the micro-benchmark. Both variables are of type T, with $T \in \{int, long, float, double\}$. In addition, we used proper default values for the second variable (i.e. using suffixes/floating point literals) to avoid implicit upcasting/downcasting operations. Since integer operations are more efficient than floating point operations due to the greater inherent computational complexity of the later, we aim at quantifying the reduced energy consumption.

3.5 Exception Handling (EH)

Exceptions are used to manage any unexpected event in the code, while ensuring code readability. When an object is in a condition it cannot handle, it raises an exception to be captured by another object. The Java Virtual Machine (JVM) searches backward through the call stack to find methods that do can handle the exception. Sadly, exception handling is expensive and involves object creation. Then, we analyze two equivalent approaches to trigger error or exceptional situations: one using exceptions and one without these to increase energy efficiency. The tested code checks whether a numeric parameter is even and if so it always raises an exception in the inefficient version of the code, and always returned a value indicating the situation in the efficient version. In practice, the second approach implies e.g. returning an error code, an error message or an invalid value, which is a simple task for programmers. The first approach intuitively is less efficient, but the goal of the experiment is to quantify how much can be reduced by employing the second approach.

3.6 Object Field Access (OFA)

Classes comprise attributes/fields, and methods with behavior. The OO paradigm encourages information hiding, so each class should provide special public methods (accessors) used by other classes to access fields in the declaring class. However, invoking accessors also has a negative impact on performance and clearly consumes energy. For our purposes we measured the energy consumption to obtain a non-static attribute value, which in one case is performed through a method call, and in the other is performed directly, i.e., without having accessors.

3.7 Object Creation (OC)

Object creation is inherent to OO because different entities with different states coexist in memory at runtime, but this involves some computational –and hence energy– cost. However, sometimes developers can avoid creating new objects of the same class by reusing objects of that class no longer used after resetting their attributes.

We analyze the impact of object creation versus reuse on energy consumption. In other words, this means creating a new instance of an application class each time it is needed, or reusing the same instance while resetting its internal state. As the possibilities to evaluate this aspect are quite diverse because of the different classes and reset behaviors that could be implemented, we chose Lists, which are often used in applications to store data in memory and are constituting parts of other data structures. Particularly, we compare the energy consumption of creating a new list (specifically ArrayList) object and insert a String into it, versus creating an instance of ArrayList once, adding the String and using the *clear()* method to reset the list instance to its empty state.

3.8 Use of Primitive Data Types (PDT)

Past programming languages only had primitive data types (integers, booleans and strings) and procedures. Developers could define their own procedures and chain them to build larger programs based on primitives data types only, but abstract types appeared later. Java has classical primitive data types that are not classes per se, but in addition each of them has a corresponding object data type (e.g., int → Integer). We then evaluate the energy consumption using primitive data types versus using object data types. For this, we test the common behavior of accumulating several values (primitive long values) into a variable V. In one case, V is of type Long, and in another case V is defined as long.

3.9 Energy-efficient Micro-benchmarks: Test Applications

We also studied savings when refactoring real-world scientific applications based on the energy-efficient versions of the micro-benchmarks. The source code was modified considering our energy-driven optimizations only, to avoid introducing potential bias

due to unintentional inclusion of other optimizations that might also contribute to further reduce energy (e.g. removing program console output). Specifically, we refactored code by just removing all occurrences of the less-efficient micro-benchmarks to apply the most efficient ones instead, which implied for example removing all exceptions and use return values in methods, resetting the same object state rather than creating a new instance each time, using primitives data types instead of wrapper classes, and so forth.

We framed our application selection based on the Phil Colella's categorization [12, 4], who identifies a set of scientific application kernels which form the basis for most existing scientific applications. We also included Machine Learning (ML) algorithms since they are widely used in a broad range of areas, such as Bioinformatics, Natural Language Recognition and Economics. To select actual projects implementing these applications, we analyzed several sources: the Ibis/Satin parallel middleware [22], the GitHub code repository and the Weka ML library [18].

From GitHub we used JAligner[2] and gradient-descent. This later is no longer available at GitHub at the time of writing this paper, and due to licencing issues, only the binary version of gradient-descent is provided by us together with the software for reproducing our experiments. From Weka we used the Bayes Network Classifier. Lastly, another four applications were extracted from the Ibis/Satin middleware.

3.9.1 Scientific Application Kernels (SFA)

Broadly, SFAs are a set of patterns that can represent broad types of scientific applications. They are in general very CPU-intensive and use primitive data structures, such as arrays and matrices.

Phil Colella's work [12] identifies a list of seven high-level numerical methods (*dwarfs*) that represent the majority of HPC science and engineering applications, and have persisted over time. That list was enlarged in [4] to consider 6 new SFAs. To both cover some of the SFAs from [12] and [4] via applications that might benefit from as many of the micro-benchmark groups explained above as possible, we using the following concrete applications:

1. Fast Fourier Transform (FFT), which can be categorized as Spectral Methods [12]. Spectral Methods are a set of techniques to solve certain differential equations, and for that purpose they use FFT.
2. Matrix Multiplication (MMult): [4] this SFA is considered as Dense Linear Algebra one, level 3 (matrix-matrix operations). These SFAs often include access to all the elements of the data structures.
3. Knapsack (KP): This problem lays in the Backtracking and Branch & Bound category since this is a combinatorial optimization problem. Backtracking and Branch & Bound SFAs are used in Integer Linear Programming and Boolean Satisfiability as well.
4. N-Queens (NQ): This problem is one of the most characteristic type of problems found in Backtracking and Branch & Bound. It solution involves using a

[2] JAligner Web page: https://github.com/ahmedmoustafa/JAligner

modified version of Backtracking to place the queens in the different possible positions of a board.

5. Sequence Alignment (SA): This is an algorithm used to align two DNA sequences in order to analyze their similitude. To this end, Sequence Alignment algorithms usually rely on Dynamic Programming.

Table 1. Test applications. Columns are AC (Array copying), MT (Matrix traversal), SH (String handling), AO (Use of arithmetic operations), EH (Exception handling), OFA (Object field access), OC (Object creation) and PDT (Use of primitive data types)

Application	AC	MT	SH	AO	EH	OFA	OC	PDT
FFT (Fast Fourier Transform)	-	-	Yes	Yes	Yes	Yes	Yes	Yes
MMult (Matrix Multiplication)	-	Yes	-	Yes	Yes	-	Yes	Yes
KP (Knapsack)	Yes	Yes	-	Yes	Yes	-	Yes	Yes
NQ (N-Queens)	-	Yes	-	Yes	Yes	-	-	Yes
SA (Sequence Alignment)	-	Yes	-	Yes	-	Yes	Yes	Yes

Table 1 summarizes the characteristics of these applications. The first column lists the test applications, while the rest of the columns are AC (Array copying), MT (Matrix traversal), SH (String handling), AO (Use of arithmetic operations), EH (Exception handling), OFA (Object field access), OC (Object creation) and PDT (Use of primitive data types). The cells indicate whether each micro-benchmarks group was present ("Yes") or not ("-") in the various applications, and hence whether the associated energy-aware refactoring opportunities apply or not. The extent to which each application uses each micro-benchmarks group naturally varies across applications. For example, FFT instantiates more objects at runtime than the rest of the applications. Applications on the other hand do not contain many input/output operations (disk usage) that might introduce noise in the energy measurements.

FFT. It computes the discrete Fourier transform, which has an impact on different areas such as image (JPEG) and audio (MP3) processing, reduction of noise in signals, analysis of frequency of discrete signals, among others. Being x_0, x_1, ..., x_{n-1} complex numbers, directly evaluating the well-known discrete Fourier transform (DFT) formula requires $O(n^2)$ arithmetic operations. However, Gauss proposed a method that requires O(n log n) steps to evaluate it, called FFT.

The algorithm in this paper is a recursive decomposition of the FFT in simple functions until obtaining 2-element functions with k={0 or 1}. Once these simple transforms are solved, the algorithm groups them in other top level computations to be solved again until the highest recursive level is reached. Lastly, the results must be reorganized obtaining the same results as the original FFT.

Mmult. It takes as parameters two matrices (A, B) containing numbers and returns another matrix (C) which holds the result of multiplying the first two matrices. Each cell c_{ij} is the addition of the products of each element in row *i* in matrix A with the corresponding element in column *j* in matrix B.

To produce the C matrix, the application used in this paper first divides each input matrix into four quadrants. This division is recursive until the last level where there is an *nxn* matrix with *n* given as a parameter. The result at any level can be computed as

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$; $C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$; $C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$; $C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$. We used *n=1* to evaluate the impact of the micro-benchmarks in the most extreme case.

KP. This is an NP-complete combinatorial optimization problem whose goal is to optimize the total value that a backpack can contain. The backpack can support a default weight *W*. The backpack is filled with elements each having a value *v* and a weight *w*. The problem arises constantly in Engineering [3] and has several applications in operation management and logistics. The version used in this paper divides the initial *N* elements into two subproblems recursively for *N−1* elements, one with the lost item placed in the backpack, and the other without it. This runs recursively until the backpack is full or there are not elements left.

NQ. Implements a classic NP-hard problem where *n* queens are placed on a NxN board so that queens can be attacked considering the chess rules. The problem has been broadly used as part of more complex applications such as OS deadlock prevention and register allocation, traffic control, robot placement for maximum sensor coverage, and many others. N-Queens is also used in many other Physics, Computer Science and industrial applications [39]. The variant used in this paper searches for every possible solution, so it is very CPU intensive.

SA. Given two DNA sequences identifies the similarity regions. A sequence is represented by a string of characters, being each a *residue*. If two DNA sequences are arranged next to one another and their most similar elements juxtapose, they are aligned. There are two types of alignment methods: global and local. The former performs the alignment of all the residues of every sequence at the same time. The local approach looks into some parts of each sequence and compares them with one part of the other. This paper focuses on the Smith-Waterman [40] local alignment algorithm, which is based on dynamic programming.

3.9.2 Machine Learning Algorithms

Machine Learning (ML) involves algorithms to allow the computer to “learn”. They take as input a structured dataset, with several properties (features) to build a model able to make estimations for new data. Supervised ML algorithms are designed for datasets where each entry has associated a set of feature values and an output –usually a category. Supervised algorithms can be further divided into classification algorithms, which target discrete outputs, and regression algorithms, which target continuous outputs. Unsupervised algorithms are applied in datasets with features data but no output. Their purpose is to find relationships among the data and split it into different cohesive groups.

ML algorithms are CPU-intensive, and may take a long time to come up with a model. In addition, they are usually modeled with matrices, and lots of operations are done with those matrices. Particularly, we will study with Gradient Descent and Bayes Network Classifier. The first algorithm is the basis for many other ML algorithms and can be categorized as *Dense Linear Algebra* according to [12]. Bayes Network Classifier is a classification algorithm that uses the Bayes theorem as the basis to build the model, and is classified as *Construct Graphical Models* according [4].

Table 2. ML applications.

Application	AC	MT	SH	AO	EH	OFA	OC	PDT
Bayes	Yes	Yes	-	Yes	-	Yes	Yes	Yes
GD (Gradient Descent)	-	Yes	-	Yes	Yes	Yes	Yes	Yes

Table 2 summarizes the two ML applications employed. The first column lists the ML applications used, while the rest of the columns are AC (Array copying), MT (Matrix traversal), SH (String handling), AO (Use of arithmetic operations), EH (Exception handling), OFA (Object field access), OC (Object creation) and PDT (Use of primitive data types). Cell values are interpreted as those in Table 1.

Gradient Descent (GD). When dealing with several variables in a function, it is computationally expensive to determine its derivative to find the global minimum. Gradient Descent iteratively optimizes until convergence the search of the local minimum for a function based on the function's gradient. In fact, most ML algorithms base their calculations on this approach or on a modified version of it [8], such as Logistic Regression, Neural Networks and Deep Learning. There are basically three types of Gradient Descent: Batch, Stochastic and Mini-batch. The first one takes into consideration the whole dataset at each iteration. The second variant performs an update round for each data point of the dataset. This is usually much faster than Batch Gradient Descent and can also be used in online learning algorithms, but it may not converge to the local minimum every time. The third approach takes groups or batches of k data points. Thus, it takes the best of the two previous alternatives (fast convergence and good solution quality).

Bayes Network Classifier (Bayes). The Bayes Network Classifier is an ML supervised classification algorithm that takes advantage of the well-known Bayes theorem to classify instances in a dataset. The dataset is processed to learn the importance that each feature has in determining the category of an instance and thus classify unknown instances. Bayes Network classifiers are used in a wide range of areas, such as information retrieval, Bioinformatics, or image processing.

The commonest variant is the Naïve Bayes Classifier, which assumes that each feature is conditionally independent from all the other random features. This usually generates a high bias in the model and reduces effectiveness. Therefore, an alternative approach [15] considers the concept of Bayes Network, which depicts the dependencies between each feature in the model.

4. Experiments

We measured the individual impact of micro-benchmarks on energy consumption (Section "Micro-benchmarks Results") and their effect on the real code described earlier (Section "Test Application Results").

The JVM includes a dynamic compiler that optimizes the parts of a program that are most frequently used [5], and a *garbage collector*, periodically launched to free unused memory. These features introduce "noise" when profiling programs, especially when these programs perform fine-grained operations, like our micro-benchmarks do. Thus, we used Google's Caliper [16], a framework for running benchmarks that deals with these problems. This research considered Java 8.

The seven applications –SFAs– were also run in multi-thread mode. Given a single-thread SFA, its multi-thread counterpart was obtained by creating several instances of the SFA in a black-box fashion, one per available core in the host computer. This was done using the Executor support of Java. For the sake of uniformity, each instance was parametrized with the same parameters as the single-thread version (primitive values or object instances depending on the case). Measuring the energy consumption of the applications running in parallel would show whether there is a relationship between energy consumption and either exploiting one CPU core or multiple CPU cores.

Note that this black-box, embarrassingly-parallel scheme to run instances of a single-thread code is actually a very popular way of conducting simulation-based experiments among scientists and engineers [46]. Many of such simulations execute the same application code (e.g. a metal deformation model) in parallel with varying values for certain parameters (e.g. applied tension) resulting in different output results (e.g. did the piece broke in each case?).

With respect to quantifying energy, the PowerMeter device[3] was used. It takes 2,000 samples (voltage, amperage, active power and apparent power) per second. We plugged a host computer –4-core AMD A8-5600K APU processor (running @3600 MHz), 8 GB RAM DDR3 and Ubuntu 17.04– to the device, which was in turn plugged to the power line. The computer connects to the device via a MODBUS RS232 port. Note that this setting means that the device cannot differentiate how much how power is consumed by a given experiment and the bare system (i.e. the software which runs when the computer is idle, mainly the operating system). In consequence, the power measured in an experiment corresponds to the whole system (computer). To quantify as accurately as possible the impact of the reduced power consumption introduced by refactoring code, we aimed to reduce the consumption levels of the computer by turning off both the network card and the screen in the computer. Running an application involved several *iterations*, for the sake of decreasing statistical errors. Upon executing an iteration, we force the application to wait until the JVM is warmed up, i.e., the state at which necessary data structures, user-level threads and internal JVM threads have been initialized. We chose *iterations* = 10, which yielded deviations < 2% for all tests.

In addition, we noted that some readings from PowerMeter were invalid (i.e., apparent power was close to 2^16), so proper support was included in our experimentation software to discard such readings. Given an individual measurement log, which therefore has stored measures corresponding to the iterations of an application, only the lines having invalid apparent power values were deemed inconsistent and hence not considered upon processing the active power readings from the log. This could be done since the standard deviation of the remaining (valid) lines was, in terms of active power, below 2%, as explained above. These actions, together with the use of Caliper, allowed us to obtain correct and usable measurements.

The experimentation software (mainly bash scripts and to a lesser extent Python code), the code itself to talk to the measurement device (written in C), and the source/binary code used in the experiments are available at a GitHub repository[4].

[3] PowerMeter Web page: http://www.powermeter.com.ar/eco/
[4] https://github.com/cmateos/Experiments-ComSIS-2019

4.1 Micro-benchmarks Results

Table 3 depicts the average power consumption (in Ws) of each micro-benchmark version. Table 4 depicts the same for the Use of arithmetic operations micro-benchmark. Within each micro-benchmark least to most efficient versions are ordered from top to bottom. EnergyUsageReduction per micro-benchmark was defined as:

$$\frac{\left[\frac{\sum Ws(exec_i) - \sum Ws(improvedExec_i)}{\sum Ws(exec_i)}\right]}{\#\ of\ iterations} * 100 \quad (1)$$

where $Ws(exec_i)$ is the consumption of iteration i of the original version of a micro-benchmark, and $Ws(improvedExec_i)$ is the consumption of an individual iteration of an improved micro-benchmark version. Ws consumed by an individual iteration is the sustained active power (in Watts) as measured from the power device considering valid readings, multiplied by the time it takes to execute the iteration (in seconds). Since the power device outputs a line of data every second, the sustained active power is the average power measured during the iteration, which was possible to use as a meaningful statistical indicator since as explained low deviations were observed even discarding the invalid readings in each iteration. In the formula, we sum up all the Ws values and then divide by the number of iterations since clearly such values might be different between individual iterations.

Array copying. To compare the efficiency of System.arraycopy we used a manual implementation of the same functionality with an array of 8KB, i.e., the default internal array size in Java for buffered readers, which are extensively used for data streams. The built-in implementation reduces energy consumption by a 37.9%. These results are in line with previous studies on Java optimization [42], where using the System.arraycopy function instead of manual array copy for the entire Java I/O piped stream subsystem resulted in likewise performance gains.

Table 3. Micro-benchmarks results (*Use of arithmetic operations* not included)

Micro-benchmark	Version	Consumption (Ws)	Energy reduction (%)
Array copying	Manual array copy	102.8	
	System array copy	63.8	37.9
Matrix iteration	By-column iteration	53,776.8	
	By-row iteration	102.6	99.8
String handling	String concatenation (+)	4,456.1	
	String builder	271.7	93.9
Exception	Use Exception	14,108.6	
handling	No Exception	28.1	99.8
Object field access	Accessor-based access	9,190.0	
	Direct access	1,700.8	81.4
Object creation	On-demand creation	813.1	
	Object reuse	461.6	43.2
Use of primitive	Use of object data types	3,082.3	
data types	Use of primitive data types	2,356.2	23.5

Table 4. *Use of arithmetic operations* micro-benchmark results

Version	Consumption (Ws)	Energy reduction (%)		
		Versus double	Versus float	Versus long
Add constant to double	5,152.5	-	-	-
Add constant to float	5,089.3	1.2	-	-
Add constant to long	3,643.5	29.2	28.4	-
Add constant to int	838.8	83.7	83.5	76.9

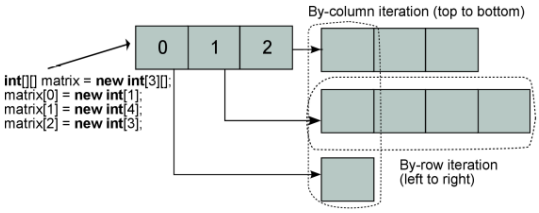

Fig 1. Two-dimensional array representation and traversing in Java

At the JVM level, using manual array copy implies copying array elements one by one, whereas invoking System.arraycopy delegates the copy to a native method. A native method be implemented differently by each JVM runtime and can be optimized in several ways that are not a possibility for Java developers. For example, the copy of the array can be done with a single *memcpy/memmove* low-level primitive from a native method, instead of *n* distinct copy operations.

Matrix traversal. This paper uses NxM matrix structures and compares traverse by rows versus traverse by columns. Specifically, a matrix of 1024x1024 was used to run tests. A key advantage of these micro-benchmarks is the simplicity of changing the traverse mode in an existing code. The results show an improvement (energy reduction) of 99.8% using the traverse by row version.

Java represents two-dimensional matrices via an array, where each cell points to another object array (Fig. 1). Overall, when a matrix is traversed by row, all the cells of arr [0] are traversed first, continuing with arr[1] and so on. When reading arr[0][0], the CPU caches the cells that are close by (arr[0][0] to arr[0][n] and may cache some cells from the next row). When the matrix is traversed by row, the next cell (arr[0][1]) is likely cached, which is faster than fetching the cell from main memory. But, when traversing by column, some of the next cell accesses (arr[1][0], arr[2][0], ..., arr[n][0]) are likely to cause a cache miss.

String handling. Table 3 shows that using the class StringBuilder directly instead of the "+" operator yields a very good improvement (1,000 concatenations were used). String literals in Java are instances of String, which are immutable meaning that their characters cannot be changed after created. Using the "+" operator involves the creation of a StringBuilder object that maintains a single internal *mutable* array of characters. Besides, the method using "+" also instantiates the StringBuilder class to handle concatenation, but performs four method calls whereas the efficient version performs three method calls.

Use of arithmetic operations. This micro-benchmark group, whose results are shown in Table 4, involved adding a constant value *c* to a numerical variable declared by varying their data type. Specifically, we resolved $X + c$ using float, double, int and long variables and constants. As a result, using the float, long and int data types yields a reduction of 1.2%, 29.2% and 83.7% respectively over relying on the double data type.

Then, double and long data types consume more energy than float and int data types, respectively, because the former provide greater accuracy and larger range of values. This means more bits to represent values and therefore more processing time. In practice, programmers should of course to keep accuracy and precision as low as possible for numerical data types in order to reduce energy consumption while not compromising the semantics of the whole application.

Exception handling. The results in Table 3 confirm that energy can be saved by avoiding exceptions. The creation of objects and the limited optimizations to the exception mechanism made by the JVM, produce higher energy consumption. To ensure minimum consumption, exceptions must be reserved only for error situations where cannot be dealt with other mechanism, for example when using third-party libraries within the application code that are designed to communicate error situations via exceptions.

An operation that includes an exception throwing executes the same lines as the same operation without exceptions but it also adds an object creation and new JVM instructions processing. Developers should define error statuses instead of using exceptions whenever possible to deal with abnormal execution flows.

Object field access. Directly reading a frequently-accessed class field yields an improvement (81.4%) because the accessor method invocation is avoided. Despite this, programmers must determine to what extent it is valuable to violate object encapsulation to favour energy efficiency. However, there are common cases in which encapsulation is not affected and energy can be reduced, e.g., accessing a class field directly from the same class or inner classes.

Object creation. By reusing objects an energy reduction of 43.2% was obtained. At the JVM level, the cost to create a new object is usually higher than the cost necessary to reset an already created object. In particular, reusing an instance of ArrayList only involves invoking its *clear()* method. This latter is efficiently implemented by just zeroing the head pointer in the internal array.

This result means developers concerned with minimizing energy consumption should not create objects arbitrarily in the code but reuse instances whenever convenient. However, energy reductions may vary depending on the objects to create: those with costly "reset" methods could outweight the benefit. In these cases, a deeper pros-cons analysis is necessary. Indeed, when running the same micro-benchmark by using Vector and LinkedList, which together with ArrayList are three of the most popular linear data structures in Java, the gain of the performed refactoring for Vector is very close to that of using ArrayList, but the refactoring increments energy usage by 1% when using LinkedList.

Use of primitive data types. The use of primitive data types yielded an energy saving of up to 23.5%. If primitive data types are used, the creation of new objects by the JVM to maintain object types is avoided. Indeed, in the previous micro-benchmark, it was shown that object creation leads to higher energy consumption. In addition, extra energy is saved since autoboxing and unboxing operations are not needed when using primitive

types. Autoboxing/unboxing is the conversion by the JVM from/to primitive types to their corresponding object type.

4.2 Test Applications Results

Table 5 and Fig. 2 show the resulting energy consumption, where the reductions in % of the refactored versions according to our micro-benchmarks with respect to the original ones have been quantified as explained earlier. Multi-thread code used the 4 cores available. Next we discuss in detail the obtained results.

FFT (Fast Fourier Transform). The main refactoring on this application was the elimination of immutable classes. This was possible through the modification of a class named Complex, which was immutable in the original test application. In the new version, Complex class instances can change the values of their attributes without creating a large number of immutable instances of such class. Also, the attributes precision of the Complex class (i.e. its real and imaginary part) was decreased from double to float without altering the FFT algorithm itself.

It is worth noting that by changing from double to float we are potentially losing precision. In Java, the double data type is 64-bit wide, with precision of up to 15 to 16 decimal points. The float data type is 32-bit wide, with precision of up to 6 to 7 decimal points. All in all, whether losing precision is problematic will depend on the application exploiting the FFT algorithm. For example, 32-bit precision suffices many audio processing related tasks.

Mmult (Matrix Multiplication). The main aspect to avoid in this test application was object creation. However, in this test application the instantiation of different classes (matrices) is performed at the beginning of the code. The matrix structure was redesigned decreasing the number of object creations: not using a recursive structure has the advantage of requiring fewer objects in memory.

KP (Knapsack). In this test application we reduced the number of objects in memory by a half. In the original version instances of the class OrcaRandom and Knapsack class were created, while in the refactored version only instances of Knapsack were created, which included the behavior of OrcaRandom.

NQ (N-Queens). This application is algorithmically rather simple. There is only one class which implements the algorithm itself, so the main refactoring for this particular case was to change the non-primitive data types and to avoid some object creation in very specific cases.

Table 5. Application results. From top to bottom, applications are listed in the order of Section "Energy-efficient Micro-benchmarks: Test Applications"

App.	Version	Consumption (Ws) / Time (s)		Energy usage reduction %	
		Single-thread	Multi-thread	Single-thread	Multi-thread
FFT	Original	1,784.76/ 27.9	947.57 / 8.1		
	Refactored	1,714.99 / 26.4	813.14 / 7	3.90	14.19
MMult	Original	34,315.15 / 496.2	22,692.00 / 183		
	Refactored	13,123.99 / 185.7	8,261.35 / 66	61.75	63.59
KP	Original	5,181.22 / 71.3	4,320.79 / 36		
	Refactored	104.94 / 1.5	103.41 / 1	97.97	97.61

NQ	Original	55,753.39 / 854	34,394.70 / 300		
	Refactored	40,315.14 / 605	22,374.64 / 189.5	27.69	34.95
SA	Original	40,813.78 / 613.6	61,832.88 / 680.7		
	Refactored	906.92 / 13.1	508.52 / 4.3	97.77	99.18
GD	Original	6,361.87 / 94	3,630.05 / 29.1		
	Refactored	3,131.95 / 44.8	1,920.94 / 15.5	50.76	47.08
Bayes	Original	14,566.14 / 114.1	1,1599.18 / 110.4		
	Refactored	11,733.39 / 88.8	1,0415.26 / 8.1	19.44	10.20

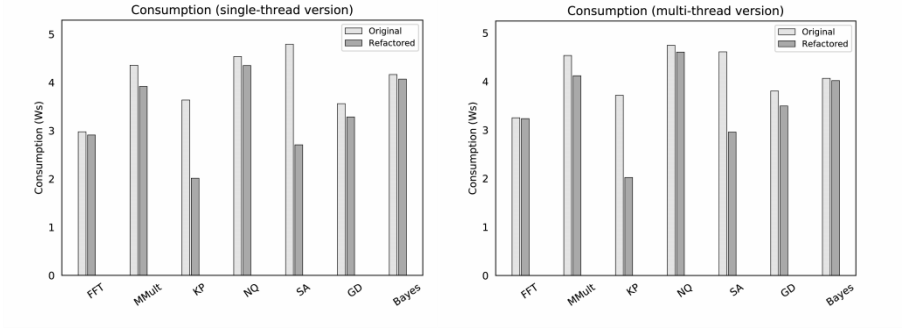

Fig 2. Consumptions for single-thread (left) and multi-thread (right) modes. Bars are log10-scaled

SA (Sequence Alignment). In the original code there is a recurrently-used class (Matrix), which is a two-dimension array of instances of the Float object type. So the most important refactoring was to use primitive data types. There were also modifications in the main class to avoid new object creations and method invocations. Note that the unrefactored multi-thread version consumed much more energy than its single-thread counterpart. As mentioned earlier, we produced multi-thread versions of applications by cloning the original application and feeding each clone with the same parameter values or instances, depending on the case. For SA, this particularly meant passing on the same object instances (two Sequence objects, representing human and mouse protein sequences), which in turn led to high memory contention among threads. However, we aimed at leaving the application code "as is" prior to refactor them and using the same multi-thread scheme for all applications, without introducing solutions to mitigate this contention. In fact, avoiding object data types and reducing object creations decreased memory usage in the refactored single-thread version.

GD (Gradient Descent). GD is a machine learning algorithm that basically learns (approximates) a multi-variable function using training data. The implementation of GD used is based on two matrices: an NxM matrix with N the number of variables and M the training set size, and another Mx1 matrix with the values of the training set. The original version of these matrixes were implemented using a Matrix class with a Collection with non-primitive data types (Double). The applied refactoring was to replace this collection with arrays of primitive data types. Thus, two further optimizations were also applied in consequence to create the optimized code. Firstly, there are less objects since one Matrix instance itself is an object.

Second, elements can be read by directly indexing an array position, i.e. without accessors. Note that this change is possible since the amount of elements in the matrices is known a priori, thus an accessor is not needed. This is possible since machine learning algorithms are usually trained with data with dimensions and sample numbers known in advance.

Bayes (Bayes Network Classifier). The implementation maps each entry of the dataset into Instance objects. Each of these objects are then processed to train the classifier. The whole set of instances (dataset) are kept in another class called Instances, which provides the methods to get or put information into it and is mainly composed by a List. In addition, similar to GD, it is possible to know the size of the dataset a priori. Thus, the refactoring applied was again eliminating the List and using an array instead.

4.2.1 Results Summary

Energy spent by an application version is computed based on active power (Watts) and runtime (seconds). For each triple T=<app, v, th>, app ∈ {FFT, MMult, KP, NQ, SA, GD, Bayes}, v ∈ {original, refactored} and th ∈ {single-thread, multi-thread}, we obtain two lists, LP and LT. LP has the active power samples from *i* iterations, and LT contains *i* elapsed times in seconds. Since our power device outputs a line of raw measurement data every one second, the size of LP is $\sum$(LT_j).

To illustrate the amount of samples in the lists, please refer to Table 5. The triple $T_{GD,o,s}$=<GD, 'original', single-thread> took 94 seconds to execute in average. LP($T_{GD,o,s}$) will then have approximately 94*10=940 samples (recall we used *i*=10 in all experiments). On the other hand, the triple $T_{GD,r,s}$=<GD, 'refactored', single-thread> took 44.8 seconds to execute in average, so LP($T_{GD,r,s}$) will have around 440 samples. Lastly, both LT($T_{GD,o,s}$) and LT($T_{GD,r,s}$) will have 10 elements, one per iteration.

We studied the source of energy reductions by performing statistical tests given T_1=<app, 'original', th> and T_2 =<app, 'refactored', th>. This means determining whether there are statistically significant differences between samples of LP(T_1) versus that of LP(T_2), and samples of LT(T_1) versus that of LT(T_2).

For energy samples, we took the active power samples lists LP(T_1) and LP(T_2) and since the lists might differ in length we run the two-tailed Mann-Whitney-Wilcoxon for unpaired data. This difference in length stems from the fact that $\sum$($LT(T_1)_j$ is usually different than $\sum$($LT(T_2)_j$, and hence the sizes of LP(T_1) and LP(T_2) also differ. For instance, the size of LP($T_{GD,o,s}$) and LP($T_{GD,r,s}$) is 940 and 440, respectively.

For elapsed times, and since the lists LT(T_1) and LT(T_2) have the same length and samples differ from each other in that a *treatment* (refactoring) is applied, we used the two-tailed Wilcoxon test for paired/matched data. This resembles the kind of test often applied on the same subject –in our case application- before and after a treatment has been applied. This is, before the treatment is applied, the application code is the original one, while after the treatment is applied, the code has been refactored. Note that each element in LT(T_1) and LT(T_2) are sampled independently, but for the sake of the statistical test they are matched, which means that the Wilcoxon test uses as input a single list with the element-wise difference of both lists.

Table 6 shows the test outcomes. Since refactored code (T_2) tended to demand more active power but less time to run than original code (T_1), we in fact tested the

significance of active power increment and elapsed time decrement of the refactored code over the original code.

Table 6. Active power and elapsed time differences: Statistical significance test outcomes (Y = Yes, N = No)

App.	Original vs refactored round	Active power decrement		Elapsed time decrement	
		At 0.01?	At 0.05?	At 0.01?	At 0.05?
FFT	Single-thread / Multi-thread	Y / N	Y / N	Y / Y	Y / Y
MMult	Single-thread / Multi-thread	Y / N	Y / Y	Y / Y	Y / Y
KP	Single-thread / Multi-thread	N / Y	N / Y	Y / Y	Y / Y
NQ	Single-thread / Multi-thread	Y / Y	Y / Y	Y / Y	Y / Y
SA	Single-thread / Multi-thread	Y / Y	Y / Y	Y / Y	Y / Y
GD	Single-thread / Multi-thread	Y / N	Y / N	Y / Y	Y / Y
Bayes	Single-thread / Multi-thread	Y / Y	Y / Y	Y / Y	Y / Y

Table 5 shows that, considering single-thread code runs, the refactored versions demanded more active power than the original versions (2-4%). The exception to this is KP, whose refactored version had 3.72% less active power. For multi-thread code, this overall trend does not hold and in fact refactored versions introduced average active power reductions compared to original code in four cases, i.e., 0.70% (FFT), 13.83% (KP), 23.18% (SA) and 0.65% (GD), which are statistically significant at the 0.01 and 0.05 confidence levels.

Another observation is that multi-thread code used more active power (between 90.83 Watts and 125.17 Watts) than single-thread code (between 63.36 Watts and 72.66 Watts). Since *Energy=ActivePower* * *RunTime*, these results show that the studied micro-benchmarks do not reduce *Energy* as a side product of *Runtime* only, but also *ActivePower* is altered.

Table 6 shows that all significant tests regarding elapsed time confirm that refactored code run faster than original code. Let us measure such improvements using the well-known speedup metric, which is the ratio between the time it takes to run an unoptimized code versus the time to run its optimized counterpart, i.e. original times over refactored times in our case. Speedups values ranged from [1.05-47.53] (single-thread) and [1.15-158.30] (multi-thread). Overall, we obtained per-iteration absolute average energy savings of 69 Ws to 39900 Ws (single-thread) and in the range of 134 Ws to 61300 Ws (multi-thread). Even when multi-thread refactored code naturally consumes more Active Power than single-thread refactored code, in the former case each core runs a refactored –and hence rather faster– version of the original code. Again, since *Energy=ActivePower* * *RunTime* the multiplicative, beneficial effect on energy consumption of using many threads can be also appreciated.

To put these savings in context, virtualization technologies –particularly Xen and KVM– and container technologies –particularly LXC and Docker– consume between 126 and 128 Ws to run eight simultaneous idle virtual guests [30]. Likewise, the energy to send 27 MB of data via TCP in metropolitan-area networks where round-trip time is up to 50 milliseconds ranges from 921 to 43000 Ws [21]. Lastly, 30000 Ws is the energy necessary to execute Kmeans clustering algorithm from the benchmark in [9] by splitting the work to do under a 50-50 scheme between a CPU and an Nvidia GeForce 8800 GTX GPU [23].

To conclude our analysis, we should also mention that the potential energy savings in an application is only an angle from which to evaluate whether it is convenient to refactor the application code or not regarding some micro-benchmarks. This way, another important angle is *analysis scope*, which refers to the quantity of code units that users have to analyze to determine where to apply refactorings without affecting the application functionality, and hence it is a qualitative measure of refactoring difficulty. This analysis might involve looking only the sections of the code where the refactoring opportunities appear, or additionally more elements like methods that call those sections or other classes. The analysis scope can be at the Statement, Method or Application levels. The Statement level particularly requires less effort from the user. For example, when refactoring for the OFA micro-benchmark, users have to change all Getter method calls by direct accesses to involved attributes (Statement level). For the MT micro-benchmark, changing the traverse orientation is a trivial task in terms of code, but it is not a trivial task at the time of analyzing the semantic of the traverse. This involves looking the method implementing the algorithm where the traverse is performed (Method level). For example, the traverse in a matrix multiplication code cannot be changed. However, after an analysis, developers could transpose the matrices and, then, change the traverse. Finally, refactoring for the AO micro-benchmark clearly implies to analyze the feasibility of reducing data types precision at the Application level.

Table 7 summarizes the micro-benchmarks based on these two angles. We have considered a qualitative indication of the energy savings that can be obtained from each micro-benchmark. In practice, this represents a prioritization for users willing to exploit our micro-benchmarks, since those yielding the best energy savings and being the most easy to apply in the code should be tackled first (e.g. OFA, EH, MT and PDT).

Table 7. Studied micro-benchmarks: energy savings and analysis scope difficulty

Micro-benchmark	Energy savings	Application scope
Array copying (AC)	Good	Application
Matrix iteration (MT)	Excellent	Method
String handling (SH)	Excellent	Application
Use of arithmetic operations (AO)	Very low-very good	Application
Exception handling (EH)	Excellent	Method
Object field access (OFA)	Very good	Statement
Object creation (OC)	Good	Application
Use of primitive data types (PDT)	Good	Statement

5. Conclusions

We have empirically assessed the energy impact of energy-friendly versions of common primitives in Java scientific code. We also show that refactoring code driven by such energy-friendly versions yield energy gains both for single-thread and multi-thread refactored applications. This gives Java scientific developers hints to build energy-efficient software for servers, which complements energy-aware approaches already proposed at the platform and hardware levels.

It is worth noting that our research benefits end user scientific applications, i.e. software whose primary purpose is not to be heavily reused (as opposed to software

libraries). In practice, refactoring an application would essentially mean modifying the original code and then properly testing the refactored code to avoid introducing bugs. However, modifying code that is aimed at being reused from other applications requires a wider view upon refactoring code to avoid breaking clients.

Consequently, if we analyze the potential impact of micro-benchmarks driven refactoring in software that is aimed at being reused, they can be grouped into those that are harmless and those that might break the software. In the former group is Array copying, Matrix iteration and String handling. Refactoring based on these micro-benchmarks means changing the way certain tasks are implemented, but software design is not broken.

Contrarily, the micro-benchmarks in the second group, i.e. the rest, might break the software design. In many cases, the library interface is affected thus breaking clients (Exception handling, Object field access, Use of primitive data types), internal object states might be violated or made inconsistent (Object creation) or what the client expects from the library might be semantically altered (Use of arithmetic operations). This does not mean our micro-benchmarks cannot be applied in libraries as well, since they would be applicable in libraries where a clear, defined separation between interface (API) and implementation exists. In this way, refactorings could be applied in principle within the boundaries of the library implementation while ensuring that the API is left untouched (both syntactically and semantically).

Finally, future work will investigate how to automatically preprocess existing code to exploit our findings. For some micro-benchmarks (e.g., object field access) this is trivial but for others (e.g., reusing objects) modification/recognition is highly challenging. We are also exploiting these ideas for mobile device programming. Preliminary works studied the rate at which micro-benchmarks versions deplete batteries [36] and the trade-off between code smell-free OO designs versus the inherent energy costs [37] in Java-based Android applications. The motivation of these works is that mobile devices can act as resource providers in edge environments to run scientific applications [20], so coding energy-aware tasks becomes crucial. In addition, we will test other common situations not covered by the micro-benchmarks code utilized in this paper. For example, these include other arithmetic operations (AO micro-benchmark), checking if a method return value is correct as opposed to having an exception (EH micro-benchmark), accessing static versus non-static object attributes (OFA micro-benchmark) and exclusively using wrapper classes in an application since boxing is avoided (PDT micro-benchmark).

Acknowledgements. We thank the anonymous reviewers for their comments to improve the paper. We acknowledge the financial support by ANPCyT through grant PICT no. PICT-2012-0045 and CONICET through grant PIP no. 11220170100490CO.

References

1. S. K. Abd, S. Al-Haddad, F. Hashim, A. B. Abdullah, S. Yussof, An effective approach for managing power consumption in cloud computing infrastructure, Journal of Computational Science 21 (2017) 349–360.

2. L. Ardito, M. Morisio, Green it available data and guidelines for reducing energy consumption in it systems, Sustainable Computing: Informatics and Systems 4 (1) (2014) 24–32.
3. J. Zhang, Comparative study of several intelligent algorithms for knapsack problem, Procedia Environmental Sciences 11 (2011) 163–168.
4. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al., The landscape of parallel computing research: A view from berkeley, Tech. rep., University of California (2006).
5. A. Barisone, F. Bellotti, R. Berta, A. De Gloria, Jsbricks: a suite of microbenchmarks for the evaluation of java as a scientific execution environment, Future Generation Computer Systems 18 (2001) 293–306.
6. R. Basmadjian, P. Bouvry, G. Da Costa, L. Gyarmati, D. Kliazovich, S. Lafond, L. Lefevre, H. De, J.-M. P. Meer, R. Pries, J. Torres, T. Trinh, S. Khan, Green data centers, Large-Scale Distributed Systems and Energy Efficiency: A Holistic View (2015) 159–196.
7. J. Brożyna, G. Mentel, B. Szetela, Renevable energy and economic development in the european union, Acta Polytechnica Hungarica 14 (7) 11-34.
8. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, G. Hullender, Learning to rank using gradient descent, in: 22nd International Conference on Machine learning, ACM, 2005.
9. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IEEE International Symposium on Workload Characterization, IEEE, 2009.
10. H. Chen, Y. Li, W. Shi, Fine-grained power management using process-level profiling, Sustainable Computing: Informatics and Systems 2 (1) (2012) 33–42.
11. A. S. Christensen, A. Moller, M. I. Schwartzbach, Precise analysis of string expressions, in: 10th International Static Analysis Symposium, 2003.
12. P. Colella, Defining software requirements for scientific computing, Tech. rep., DARPA's High Productivity Computing Systems (HPCS) (2004).
13. G. Dhaka, P. Singh, An empirical investigation into code smell elimination sequences for energy efficient software, in: 23rd Asia-Pacific Software Engineering Conference, 2016.
14. European Commission, Code of conduct on data centres energy efficiency, Tech. rep., Institute for Energy, Renewable Energies Unit, v2.0 (2009).
15. N. Friedman, D. Geiger, M. Goldszmidt, Bayesian network classifiers, Machine learning 29 (2-3) (1997) 131–163.
16. Google, Caliper, https://github.com/google/caliper/wiki/ProjectHome.
17. A. Greenberg, J. Hamilton, D. A. Maltz, P. Patel, The cost of a cloud: research problems in data center networks, ACM SIGCOMM Computer Communication Review 39 (1) (2008) 68–73.
18. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The weka data mining software: an update, ACM SIGKDD explorations newsletter 11 (1) (2009) 10–18.
19. S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: 38th International Conference on Software Engineering, 2016.
20. M. Hirsch, J. M. Rodriguez, A. Zunino, C. Mateos, Battery-aware centralized schedulers for cpu-bound jobs in mobile grids, Pervasive and Mobile Computing 29 (2016) 73–94.
21. M. Usman, D. Kliazovich, F. Granelli, P. Bouvry, P. Castoldi, Energy efficiency of tcp: An analytical model and its application to reduce energy consumption of the most diffused transport protocol, International Journal of Communication Systems 30 (1).
22. R. V. van Nieuwpoort, G. Wrzesińska, C. J. Jacobs, H. E. Bal, Satin: A high-level and efficient grid programming model, ACM Transactions on Programming Language and Systems 32 (3) (2010) 1–39.
23. K. Ma, Y. Bai, X. Wang, W. Chen, X. Li, Energy conservation for gpu–cpu architectures with dynamic workload division and frequency scaling, Sustainable Computing: Informatics and Systems 12 (2016) 21–33.

24. I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: 36th International Conference on Software Engineering, ACM, 2014.
25. C. Mateos, A. Rodriguez, M. Longo, A. Zunino, Energy implications of common operations in resource-intensive java-based scientific applications, in: New Advances in Information Systems and Technologies, Springer, 2016, pp. 739–748.
26. L. Minas, B. Ellison, Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers, Intel Press, 2009.
27. A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444–449.
28. A. Nicolaos, K. Vasileios, A. George, M. Harris, K. Angeliki, G. Costas, A data locality methodology for matrix-matrix multiplication algorithm, Journal of Supercomputing 59 (2012) 830–851.
29. A. Noureddine, A. Bourdon, R. Rouvoy, L. Seinturier, A preliminary study of the impact of software engineering on greenit, in: 1st International Workshop on Green and Sustainable Software, 2012.
30. R. Morabito, Power Consumption of Virtualization Technologies: An Empirical Investigation, in: IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), 2015.
31. A.-C. Orgerie, M. D. d. Assuncao, L. Lefevre, A survey on techniques for improving the energy efficiency of large-scale distributed systems, ACM Computing Surveys (CSUR) 46 (4) (2014) 47.
32. S. Papadimitriou, K. Terzidis, S. Mavroudi, S. Likothanassis, Exploiting java scientific libraries with the scala language within the scalalab environment, IET Software 5 (2011) 543–551.
33. G. Pinto, F. Soares-Neto, F. Castor, Refactoring for energy efficiency: A reflection on the state of the art, in: 4th International Workshop on Green and Sustainable Software, GREENS '15, IEEE Press, 2015.
34. G. Procaccianti, H. Fernández, P. Lago, Empirical evaluation of two best practices for energy-efficient software development, Journal of Systems and Software 117 (2016) 185–198.
35. N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, M. Valero, Supercomputing with com modity cpus: Are mobile socs ready for hpc? International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013.
36. A. Rodriguez, C. Mateos, A. Zunino, Improving scientific application execution on android mobile devices via code refactorings, Software: Practice and Experience 47 (5) (2017) 763–796.
37. A. Rodriguez, C. Mateos, A. Zunino, M. Longo, An analysis of the effects of bad smell-driven refactorings in mobile applications on battery usage, in: Modern Software Engineering Methodologies for Mobile and Cloud Environments, IGI Global, 2016, pp. 155–175.
38. C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: 1st International Workshop on Green and Sustainable Software, 2012.
39. P. San Segundo, New decision rules for exact search in n-queens, Journal of Global Optimization 51 (3) (2011) 497–514.
40. T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1) (1981) 195–197.
41. G. L. Taboada, S. Ramos, R. R. Exposito, J. Tourino, R. Doallo, Java in the high performance computing arena: Research, practice and experience, Science of Computer Programming 78 (5) (2013) 425 – 444.
42. J. Zhang, J. Lee, P. K. McKinley, Optimizing the java piped i/o stream library for performance, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 2002.

43. R. Pereira, P. Simao, J. Cunha, J. Saraiva, jStanley: Placing a Green Thumb on Java Collections. 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018.
44. R. Pereira, M. Couto, J. Saraiva, J. Cunha, J. Fernandes, The Influence of the Java Collection Framework on Overall Energy Consumption. 5th International Workshop on Green and Sustainable Software, ACM, 2016.
45. R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. Fernandes, J. Saraiva, Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? 10th ACM SIGPLAN International Conference on Software Language Engineering, ACM, 2017.
46. D. Monge, E. Pacini, C. Mateos, C. García Garino. Meta-heuristic based Autoscaling of Cloud-based Parameter Sweep Experiments with Unreliable Virtual Machines Instances. Computers & Electrical Engineering - Special Issue on 7th special section on Cloud Computing 69 (2018) 364-377.

Mathias Longo holds a BSc. in Systems Engineering from the UNICEN, and he is currently pursuing an MSc. in Data Science at the University of Southern California.

Ana Rodriguez holds a BSc.in Systems Engineering from the UNICEN, and a Ph.D. in Computer Science from the UNICEN (2018), working under the supervision of Alejandro Zunino and Cristian Mateos.

Cristian Mateos holds an MSc. and a Ph.D. in Computer Science from the UNICEN. He is an adjunct professor at the UNICEN and researcher at the CONICET. He is interested in parallel and distributed programming, middlewares, and mobile/service-oriented computing.

Alejandro Zunino holds an MSc. and a Ph.D. in Computer Science from UNICEN. He is an adjunct professor at the UNICEN and researcher at the CONICET. His research areas include grid computing, service-oriented computing, Semantic Web services, and computer security.

Received: June 08, 2018; Accepted: June 01, 2019