How evolutionary algorithms consume energy depending on the language and its level

Juan J. Merelo-Guervós[1][0000−0002−1385−9741] and Mario García-Valdez[2][0000−0002−2593−1114]

[1] Department of Computer Engineering, Automatics and Robotics, University of Granada, Granada, Spain
[2] Department of Graduate Studies, National Technological Institute of Mexico, Tijuana, Mexico
jmerelo@ugr.es, mario@tectijuana.edu.mx

Abstract. Making evolutionary algorithms greener implies tackling implementation issues from different angles. Practitioners need to focus on those that can be more easily leveraged, such as the choice of the language that is going to be used; high-level (interpreted), mid-level (based on multi-platform virtual machines), and low-level (native) languages will need power in different ways, and choosing one or the other will have an impact on energy consumption. We will be looking at the implementation of key evolutionary algorithm functions, in three languages at three different levels: the high-level JavaScript, the mid-level Kotlin, which runs on the Java Virtual Machine, and the low-level Zig. Looking beyond the obvious, as the lower the level, the less energy consumption should be expected, we will try to have a more holistic view of the implementation of the algorithms in order to extract best practices regarding its green implementation.

Keywords: Green computing, metaheuristics, JavaScript, energy-aware computing, evolutionary algorithms, zig, Kotlin

1 INTRODUCTION

The interest in greener computing has grown in the last decades. In part this has been fueled by the Millennium Development Goals[3] that advocate for a lower carbon footprint in human activity, but also due to the fact that performance improvements brought by faster hardware are not coming with the same speed as they used to [19].

There are, however, no universal solutions, so we need to focus on specific field; the general field of artificial intelligence has received a lot of attention lately; we are, however, interested in evolutionary algorithms [3], which, broadly speaking, fall within that field, but, unlike it, is not targeted by specific hardware processors relying on general CPU power (and its consumption). In the general sense, there are many different ways to lower the carbon footprint of computing

[3] Described in https://www.un.org/millenniumgoals/bkgd.shtml, for instance

workloads, however: as Hidalgo et al. affirm [7], there are four different levels for the evaluation and eventual improvement of energy consumption in AI (including, as in this paper, metaheuristics): just above the hardware level there is the programming language level, which includes not only the language itself, but also libraries and compilers or interpreters (inclusively named the *toolchain*. The latter was the topic of [15], where our focus was to work on different JavaScript interpreters in order to discover which one used system energy more efficiently.

However, there are many user cases in which the language used by all or part of the metaheuristic implementation can also be chosen. A simple choice of language will greatly impact energy consumption, as shown in [17] or [12]. In [16], we looked at the performance of different languages when implementing an evolutionary algorithm; in this paper, we will focus on the language level, before delving into specific language characteristics. Languages can be classified along many different axes: compiled vs. interpreted (or compiled natively vs. complied to bytecode, as is the case with Java and Kotlin), low vs. high-level, functional vs. procedural vs. object-oriented, for instance. In most cases, it is not a dichotomy, but a continuum; most interpreted languages, for instance, use Just-In-Time (JIT) compilers before actually running a script; most languages are also, nowadays, multi-paradigm, using procedural and functional as well as object-oriented features. Interpreted vs. compiled, low-level vs. high-level are still useful distinctions, however, from the point of view of performance as well as what interests us most in this line of research, energy consumption.

Thus, for this paper, we will work with languages that occupy different positions along those two axes. We will work with a high-level language, JavaScript, using the fastest implementation available, `bun`; this is also an interpreted language, a mid-level (that is, high-level from the point of view of language design, low-level in the sense that it is compiled to bytecode), compiled language, Kotlin, that compiles to bytecode of the Java virtual machine (JVM) and a low-level language, Zig, a relatively new language that is still not reached production, but is however used for `bun` itself[4]. All these languages have toolchains that are free software and thus can be used without any kind of limitation.

These languages can also be divided along two different axes that impact on their energy consumption:

- Memory management: it is done automatically in high-level languages, like JavaScript and Kotlin[5], while it involves a series of choices in the case of Zig. Inasmuch as allocating and releasing memory consumes energy, there is going to be a difference between having a runtime or interpreter make heuristic choices, or the developers making those decisions themselves.
- Compilation: JavaScript is interpreted, Kotlin is compiled to bytecode and then interpreted by the JVM, and Zig is compiled to native code, but this also implies what happens with optimization: high-level languages take all

[4] Please note that there are no languages that are considered low-level and interpreted, although there are minimalist interpreted languages like Lua [9]

[5] Which, for the purposes of this paper, is considered mid-level, since it compiles to bytecode

decisions, while Zig will allow you to choose between different optimization levels; it also means that a different kind of overhead will be incurred when running a program: JavaScript will load the interpreter in memory, and then parse and run the program; Kotlin will actually use the Java virtual machine to run, and it will be loaded and then the bytecode parsed and run; finally, Zig creates executables, which will have some overhead due to standard linked libraries.

In general, this implies that although *a priori* we can assume that a low-level language, being *closer to the iron*, will consume fewer resources, the fact that high- or mid-level languages apply heuristics and best practices to those decisions might eventually mean that, on the default case, the balance might be tipped towards high-level languages; this is going to be the main focus of this paper.

Making comparisons of energy spent by different implementations needs, first, a methodology to make choices that are comparable across all three platforms; then what exactly is going to be measured needs to be established, and how to make those measurements so that they make differences stand out. As in previous papers, we will focus on two critical evolutionary algorithm functions [1]: the fitness function, as well as *genetic* operators. However, we will use a heavier fitness function in this case: Hierarchical If and only IF (HIFF) [21] for independent measurement, as well as combine mutation and crossover when solving the OneMax optimization problem. In general, we are going for combined, or more complex, operations in this paper so that we can implicitly evaluate more features for every language, such as function calling or argument passing, which will have a different implementation, and thus energy overhead, in every language.

The rest of the paper is organized as follows: next, we present the state of the art in software engineering and its analysis of energy consumption of different software platforms, to be followed by the methodology we are going to apply in this paper, mainly concerned with evolutionary algorithms in 3. Then we will present the results of the experiments in 4, and finally, we will discuss the results and draw some conclusions in 5.

2 State of the art

The "shade of green" of different languages has been repeatedly examined in the literature, at least since it actually became a concern in software engineering [18], very recently, indeed. This concern has been systematized in what are called the GREENER principles [11], which try to provide a foundation for Environmentally Sustainable Computer Science (ESCS), and are placed at different functional and pragmatic levels, from governance to education (that would be the last E). In this paper, we are mainly concerned with two other Es, estimation and "Energy and embodied impact," as well as the second R, Research. These principles seek to monitor and then minimize the energy needs of computations; the Research principle, in turn, encourages investigating those topics. We will

call these principles EER, for short. The GREENER principles try to kick-start a feedback loop that allows researchers, and then developers in turn, to keep making decisions so that the same wallclock performance can be obtained with a lesser amount of energy.

The enunciation of these principles is very recent; however, they have been applied to different areas, in different forms in the case of evolutionary algorithms, these have been studied for some time [20] following the EER principles' point of view; however, it is interesting to note what these studies have focused in: population size [4], hardware platform [20], the kind of algorithm [6] or the specific interpreter chosen to run the algorithm written in a specific language, JavaScript [15]. Although in this last case, in specific situations, energy savings of 90% can be achieved, there are other possible choices that, in principle, might have a more significant impact without sacrificing performance (as would be the case when working with different hardware platforms).

One of the ways of applying these EER principles is researching the programming language where we are going to implement the workload itself. A comprehensive and recent study [17], that measures energy consumption on a general workload (the so-called CLBG corpus) shows that low-level languages like C, Rust or C++ are consistently coming on top. Mid-level languages like Java are placed 5th on the overall energy ranking, with almost double energy consumption. Interpreted languages like JavaScript (probably using the mainstream interpreter, node.js) are placed 17th, spending energy at four times the rate of C. The last language in the ranking, Perl, spends two orders of magnitude more than the baseline; Python is next to last, with a less significant difference. Choosing the right platform is not everything, however, [12] shows that the same algorithm implemented using different data structures might yield different levels of consumption; a similar problem is approached in [2], that shows that using the visitor pattern has a big impact in the energy profile of any program; however, the achieved reductions are very different depending on the language: while there is a slight reduction in the case of Java, the reduction achieved is dramatic in the case of C++. This probably shows that when we are comparing platforms we need to include the evaluation of higher-level components, that is, we need to go beyond the analysis of single functions.

In this paper, however, we will focus on the choice of programming languages based on how much energy they consume when applying the main operators of evolutionary algorithms. We also consider the role of higher-level language elements in its energy consumption. How we will be doing this work is presented next.

3 Methodology

An energy profiling methodology will start with selecting a tool to perform measurements, then present the workload that is going to be used, proceed to the different systems that are going to be measured, and end with the specific

versions of the instruments and systems to be used; after this, the circumstances under which the measurement is going to take place need to be presented.

The measuring instrument was already chosen in a previous paper [15]. In general, there are system-wide as well as per-process measuring tools. We settled for pinpoint [10], a command line tool under active development that takes per-process measures that are more accurate than other system-wide tools; it is also able to work with different sensor APIs across different operating systems, giving us an uniform tool to measure energy consumption.

In that paper, we selected a single integer arithmetic fitness function, MAX-ONES, and the crossover operation. However, in this paper we are taking a wider view of the problem so we will try to work with fundamental building blocks, but with ones that involve a bigger range of runtime system of the language. We will thus use two functions:

- The Hierarchical If and only IF (HIFF) [21] is an integer arithmetic function that works recursively, reducing a Boolean string of 0 and 1s by splitting it and applying the function to the halves; the final result will depend on the organization of 0s and 1s in the initial string, but will anyway involve many recursive function calls that will have to make use of the heap.
- The other function will apply crossover to randomly selected pairs of binary strings, followed by mutation to every member of the resulting pair, and evaluate the ONEMAX function on that result. These are essentially five function calls, not as complicated as before, but it is a fundamental operation in evolutionary algorithms and will essentially test the ability to access random elements in a string, as well as the creation of new ones (or the cloning of them, depending on the implementation)

We will use a workload of the same size as in the previous paper, 40K chromosomes, with chromosomes of different sizes: 512, 1024, and 2048 bits. This is a difference with respect to the previous paper, where we did not use 512 bits, using 4096 instead. We consider that the bigger size is not as realistic, and is relatively unlikely to be found in real problems; besides, the amount of time needed to compute HIFF was extremely high for the high-level JavaScript, so we decided to drop it so that experiments can take place in a relatively reasonable amount of time.

The languages have been chosen using a specific criterion:

- JavaScript is, as shown in the state of the art, one of the languages that consume the least energy as proved in [17]; in our previous paper, we have also found that using the bun interpreter can results in savings as high as 90% [15]. It is a high-level, object-oriented language that is, indeed, quite popular; popularity is the main criterion we have used to choose languages used. It has been preferred over other languages that might have a higher popularity in metaheuristics, such as Python, since the above-mentioned paper includes it as one of the languages that consume the most energy in a general payload; metaheuristics need not be an exception. At any rate, this paper is about choosing among kinds of languages, so any result that is

obtained might be extended to languages of the same kind; at the same kind, we propose a methodology for choosing the right system for metaheuristics, so more precise measurements would have to be performed on whatever alternative to JavaScript we want to introduce. The implementation of the above-mentioned function is open source and hosted at https://github.com/JJ/energy-ga-icsoft2023 with a free license. The actual code is the same used in the above mentioned paper.

- Kotlin is a language that compiles to the JVM, and in that sense, it would be comparable to Java in terms of energy consumption. However, data structures and other runtime characteristics might differ, as well as the workload used to compare it with other languages, so what we obtain in terms of ranking might be different from what [17] shows. As we mentioned above, and on a first approximation, results obtained here might be extended to other languages such as Java, Scala, or Clojure that also target the JVM. Kotlin has not been the target of any popular EA library as far as we can tell, although it is mentioned in this context in works such as [8]. The implementation that has been used is also included in the same repository and has been adapted from the one used in [13,14]. In that study, Kotlin was one of the fastest, even faster than Java in one of the versions. Being the main language used to create Android apps, its popularity makes it meet the main criterion used to choose languages in this paper.
- Finally, we use Zig as the low-level language instead of the more popular in EA circles C++, or, in general terms, Rust. We chose it mainly because the interpreter we use for JS is written using it; so we should expect less energy consumption when we lower the level of abstraction. On the other hand, it would be a totally new implementation of evolutionary algorithms, so this work could serve to introduce the language (and its possibilities) to practitioners. Again, the implementation is hosted in the same repository as the others used for this paper. This language cannot exactly be said to be as popular as the other two in this study; however, since it is the language used to implement the JavaScript interpreter `bun`, it was precisely the lowest level of the chosen high-level language, so we were interested to see how that would translate to differences in energy consumption. Another criterion used to choose this specific one is that, being an emergent language, there are no studies that we know of that work out its energy consumption.

All experiments for this paper have been carried out in a Linux machine `5.15.0-94-generic #104 20.04.1-Ubuntu SMP` using AMD Ryzen 9 3950X 16-Core Processor. These are the versions used for every tool and language:

- `pinpoint` does not have a version, but it has been compiled from commit `1578db07b1ee30318966d7a2097ee1bb219a9dc8`, October 26 2023.
- `bun` uses version 1.0.7.[6]

[6] Please note that at the time of writing this, many other versions have been published; we have used this one to be able to compare with previous papers. These same papers show that its performance and energy consumption have important improvements,

- zig uses version 0.11.0. This version, released by August 3, 2023, is the last one at the time of writing this paper.
- Kotlin version string is `1.9.22-release-704`; this includes the JVM version, OpenJDK 11.0.21.

All programs are run through a Perl script that captures and processes output, generating CSV data files that are committed to this repository, and available under a free license. Instructions to compile and run it are also included in the repository; in general, all that is needed to reproduce the results of this paper. All the code is automatically tested and tagged so that the exact version used in this paper can be retrieved.

Implementing an algorithm will always imply some choices in the specifics used to program it. In general, we opted for the default implementation (as suggested by tutorials), but explicitly, these are the decisions we took:

- bun uses the same implementation as previously used, namely, mutable strings, to represent the chromosome.
- zig, being a low-level language, needs a bigger set of choices. We have used `DefaultPrng` as random number generator, `page_allocator` as allocator, and arrays of byte-size integers (`u8`) for the chromosomes. This is the default implementation of integers in the language. The executable has been generated with default options too (`optimize` for optimization).
- Kotlin uses again the same implementation as [13], a `BooleanArray` for every chromosome.

All results shown below are averages for 15 runs for every configuration.

Please note that no sort of energy-wise optimizations have been performed on these implementations, since the main users of this work would be students and scientists with no deep knowledge of specific programming languages who want to create energy-efficient implementations of their algorithms. Every one of the languages has different levers that could be used to optimize energy, but that is not the focus of this paper. At any rate, if there is a close call in the results, the reader should take it into account and again measure energy consumption for specific workloads.

Previously, the experiment runner eliminated the baseline energy consumption by subtracting the average consumption of an idle process during the average time the experiments took; however, that implied that the energy (and time) measured included *generation* of the chromosomes, as well as the operations themselves.

In this paper, we will factor out that time, as well as the energy spent. Since the operation of generating chromosomes is made just once at the beginning of the run in evolutionary algorithms and is thus not very significant for evolutionary algorithm as a whole[7], subtracting it from the experiment run time will

so in case of close calls, we would recommend retaking measurements at the time of running your experiments

[7] We are not saying here that it is not an energy-consuming operation, even more so here where we are generating *all* chromosomes we are using in a single loop; however,

allow us to focus on the added energy consumption of the operations themselves; the results published in the next section will then subtract the average time and energy consumption of this operation, which is shown in Figure 1.

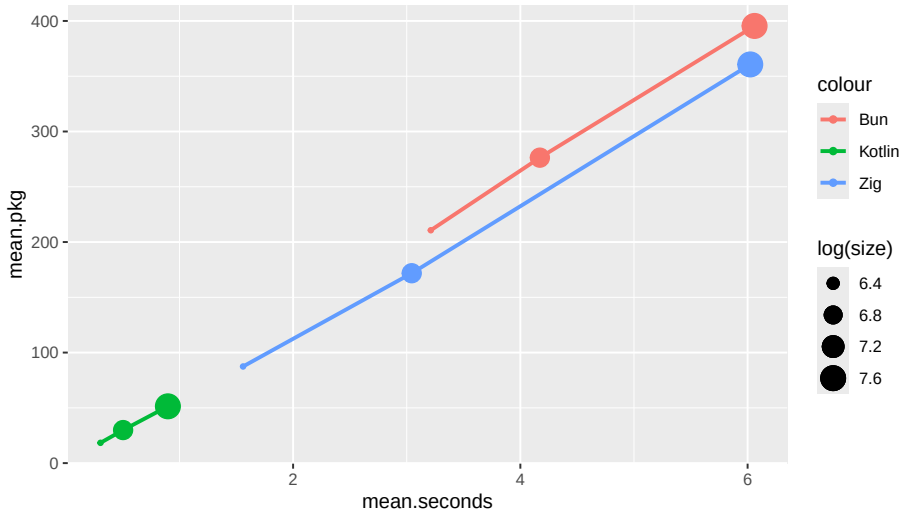

Fig. 1. Average running time and PKG (CPUs and memory) energy consumption generating 40K chromosomes for the three languages (represented with different colors); dot size is proportional to the logarithm of the chromosome size.

This Figure 1 shows average energy consumed (PKG, by CPU and memory `mean.pkg`, GPU use is negligible for this problem) vs. time taken (`mean.seconds`), already allows us at least an initial comparison of the orders of magnitude of the difference we should expect. Clearly, Kotlin is the fastest and also the one that consumes the least energy; it is followed by zig, although time as well as energy consumption grow very fast with the size of the chromosomes (here represented logarithmically as the dot size). bun is the slowest, as well as the one that consumes the most energy, although we should note that for the bigger size (2048 bits) zig takes almost the same amount of time, although with a lower consumption of energy[8].

while the initial population is generated only once in an EA, every other operation used here is going to be repeated every generation; thus, in terms of either time of energy consumption, the proportion employed by chromosome generation will be one vs. number of generations needed to find the solution

[8] zig will greatly vary the amount of energy needed depending on relatively irrelevant choices such as using constant or mutable pointers; in a previous experiment run, included in the repository, that used mutable pointers, energy consumption was almost 10% higher.

We will proceed to the experiments on an evolutionary algorithm workload, which will use this as a baseline.

4 Results

The first experiment will perform 40K crossover + mutation + onemax operations on chromosomes of size 512, 1024 and 2048.

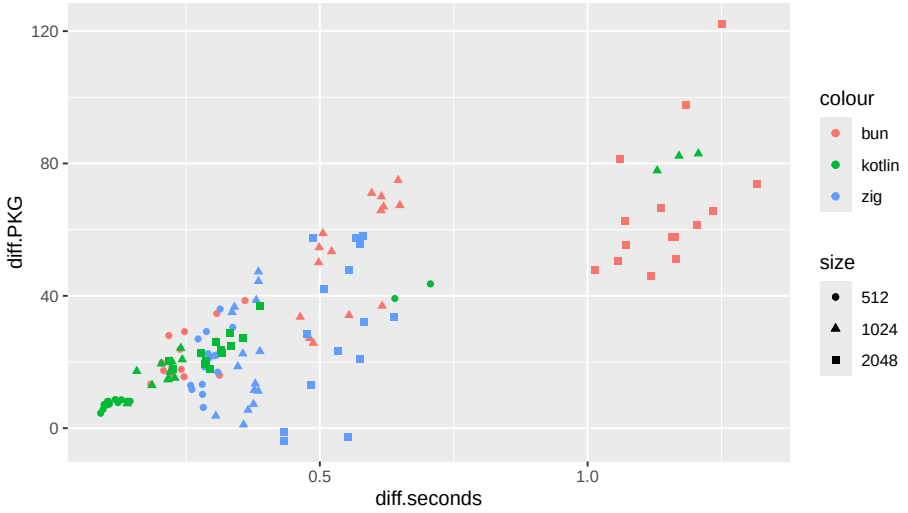

Fig. 2. Running time and PKG energy consumption processing 40K chromosomes via crossover, mutation and ONEMAX for the three languages (represented with different colors); dot shape represents the chromosome size.

Figure 2, shows the difference in running time and PKG energy consumption for the three languages examined. We have opted to show the results for every experiment in a single plot, to reveal trends as well as some quirks that might be explained more by language idiosyncrasies than by experimental errors. For instance, we can see that there are several cases where the Kotlin experimentation takes a long time, even for 1024 bits and 512 bits. One of the issues that the JVM has is garbage recollection; if there is a garbage recollection cycle it can clearly impact performance. The fact that it has hit a percentage of the experiments reveals that specific behaviors need to be taken into account and, to the extent that it is possible, mitigated.

Another quirk is the lower-than-0 energy consumption for zig, simply revealing that the difference in energy consumption is so low that it falls below the average for the generation of the chromosomes. zig's energy consumption is never higher than 60 Joules, anyway.

The trends need to be pondered, too. In general, bun is going to take longer and consume more energy than the rest for every size; Kotlin is the fastest and boasts the lower consumption of energy overall for smaller sizes. zig, on the other hand, is faster than bun, and its speed is quite consistent; not so for consumption of energy, which can go from very low, to even less than Kotlin in some experiments (and virtually zero), to relatively high, with a maximum closer to the average for bun.

Table 1. Average operations per Joule in the combined operations experiment for the three languages.

Language	Size	PKG average	PKG SD	Ops/Joule average
bun	512	22.26	7.58	1796.68
bun	1024	52.72	17.12	758.79
bun	2048	66.51	20.57	601.40
zig	512	20.01	8.34	1998.67
zig	1024	21.35	15.52	1873.42
zig	2048	30.82	22.36	1298.06
kotlin	512	12.00	12.02	3334.26
kotlin	1024	29.67	26.88	1348.22
kotlin	2048	23.71	5.10	1686.72

What we see in Table 1 is a complicated scenario, where Kotlin has a very high number of operations per Joule, with a power consumption as low as 12 Joules for the smallest size, although the trend is slightly different for the middle size of 1024 bits, where zig obtains the lowest energy consumption and operations per Joule, mainly due to the fact that the standard deviation for Kotlin, probably due to garbage recollection operations, is very high. Remarkably, the energy consumption for Kotlin can be as low as 1/3 of what bun requires; zig requires half (although they can be very close for the smallest value, where Kotlin excels).

We need to check the speed and consumption of the selected fitness function, HIFF, for the three languages, so we can have a more complete picture of the energy profile for the three languages. This is a heavy-weight function, involving function calls in the order of one thousand. Even if it is going to take much more time than generating the chromosomes, we will still subtract the time needed for that operation to compare different things uniformly.

The operation for these functions needed some tweaking, namely, conversion to string in the case of Kotlin and to constant string in the case of zig. This adds what is essentially a copying operation to the fitness itself, but we decided to do that in order to have HIFF defined in the most similar way possible (working with strings of 0s and 1s).

The results for time and energy consumption are represented in Figure 3. We can already observe that the consumption is more than one order of magnitude higher than before, and since the number of function calls is dependent on the

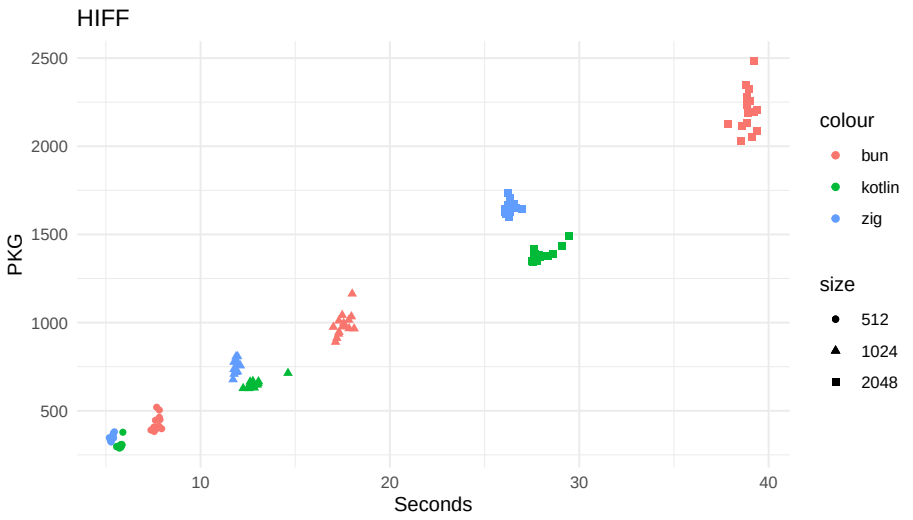

Fig. 3. Running time and PKG energy consumption computing the HIFF fitness function for 40K chromosomes for the three languages (represented with different colors); dot shape represents the chromosome size.

chromosome size, it grows more or less linearly with size. We can also observe that, as usual, bun is the slowest and the one that consumes the most energy.

However, the situation with zig and Kotlin is different. Kotlin is in most cases slightly slower, but also more energy-saving, thus more *green* than zig. We represent a boxplot comparing them in Figure 4.

Differences are significant for all three languages, for all sizes involved; in the case of the biggest size, there is indeed a considerable difference, with Kotlin spending less than 1500 Joules on average.

Table 2. Average operations per Joule in the HIFF experiment for the three languages.

Language	Size	PKG average	PKG SD	Ops/Joule average
bun	512	425.91	41.91	93.92
bun	1024	987.69	64.54	40.50
bun	2048	2204.06	123.40	18.15
zig	512	349.66	15.60	114.40
zig	1024	747.59	37.19	53.51
zig	2048	1651.50	35.18	24.22
kotlin	512	305.29	20.93	131.02
kotlin	1024	648.83	22.88	61.65
kotlin	2048	1386.58	38.43	28.85

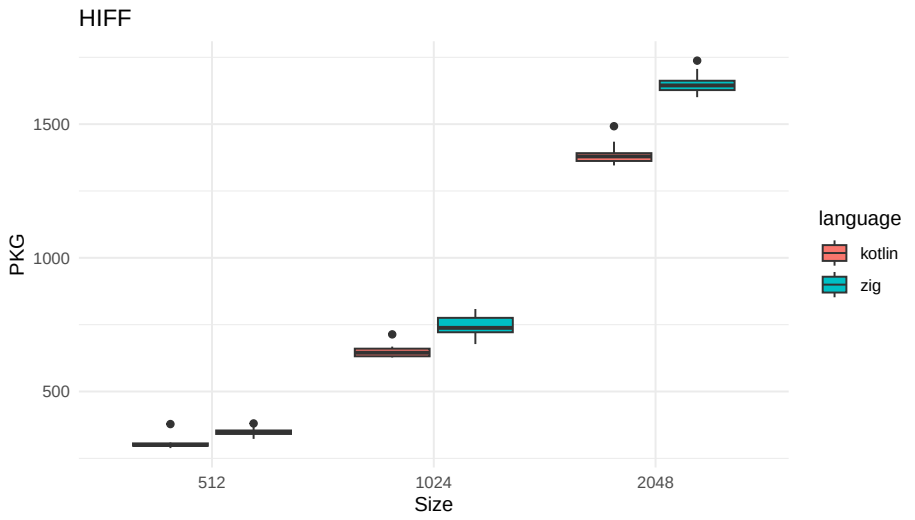

Fig. 4. Boxplot of the energy consumption of the HIFF fitness function for 40K chromosomes for Kotlin and zig

What we see in Table 2 is the number of operations per Joule all three languages are able to perform.

5 Discussion, conclusion and future work

In this paper, we have performed a series of experiments running an evolutionary algorithm workload using three different languages with different *levels* (different ways of running the –possibly compiled– code), which translate to different ways of running the algorithms. Focusing on a set of operations allows us to create specific energy profiles for the algorithm implementations, as well as the languages. The methodology followed enables to benchmark the energy consumption of the different languages, as well as to differentiate them; it does not consume an excessive amount of time. Showing the results as operations per Joule focuses the the attention on how "green" every language is, by showing precisely how many operations can be performed with a certain amount of energy.

In this work, we compared the performance regarding power consumption of three languages, JavaScript (using the bun high-performance interpreter), Kotlin (using a free JVM), and zig, when implementing certain important operations in evolutionary algorithms. We have used two workloads, one with a combination of genetic operators and the OneMax fitness function (which would be a lightweight, although widely used, combination), and another with the HIFF fitness function, which is not only heavier in terms of operations, but also recursive, thus involving specific language overheads.

In all cases (that include also the generation of the chromosomes measured, used as baseline) we have found that the combination of Kotlin and the JVM is the most energy-efficient language (as well as the faster, although that was not the main focus of the experiment), followed by zig, and finally JavaScript using bun as interpreter. The difference is bigger for core-only operations, where Kotlin can be twice as fast as bun, than in other operations that involve the overhead of function calls, like HIFF, where the difference is much smaller, around 5% for zig and Kotlin in the chromosomes with the biggest size. The fact that zig can be slightly faster than Kotlin in this last case is not relevant from the point of view of the energy efficiency; Kotlin still needs less energy even if it takes a bit longer; you can trade off easily energy efficiency for speed in this case, since the average difference is just a few percentage points.

The relatively small difference between Kotlin and zig might imply that with some engineering and optimizations, zig energy efficiency might be on a par with Kotlin. Low-level languages have many different ways of optimizing its design and tooling, and we did not create the zig program at an expert level. However, this level is consistent with the use case we are giving the application, those of a scientist acting as a developer, not an expert developer of system software (which is the most common use case for zig). We cannot then affirm that Kotlin is the most efficient across the board, but we can definitely propose it as the *greener* technology for the specific case of evolutionary algorithms that do not have an extensive amount of GPU use; this would include mainly tests for new operators or, in general, new methodologies for evolutionary algorithms that use common fitness functions.

The interesting thing about modern experimental setup is that different languages can be easily mixed in a single application. As we can see in the experiments above, energy expenses of fitness functions can be twice as high as the rest of the operations; using *green* languages exclusively in this case, leaving the rest of the application to easier-to-use interpreted languages will not have a big impact in the energy profile, while simplifying the development as well as the integration within current frameworks such as DEAP [5].

As future lines of work, we plan to extend this study to other languages and workloads, especially those that use different kind of virtual machines like Haskell or Elixir. On the algorithmic side, another research venue is to analyze the energy consumption of languages implementing different concurrency models, implemented either by language constructs or through specialized virtual machines. We also plan to analyze the energy consumption of different hardware platforms (including virtualized cloud platforms), and to develop a tool to help researchers and developers to choose the most energy-efficient language for their needs. Different fitness functions will also present a different energy profile, so including them in the investigation might help us have a more complete dashboard to be able to minimize resource consumption of EA implementations. Fitness function that employ floating-point arithmetic would need to use the GPU, with a totally different energy profile, so that will need to be taken into account.

An investigation focused on zig would also help understand how low-level languages can be made *greener* and to what extent energy efficiency can be improved.

Acknowledgements

This work is supported by the Ministerio español de Economía y Competitividad (Spanish Ministry of Competitivity and Economy) under project PID2020-115570GB-C22 (DemocratAI::UGR).

We are also very grateful to the zig community, without which programming these operations in that language would have been impossible.

Data availability

The source of this paper as well as the data and whole writing history is available from https://github.com/JJ/energy-ga-icsoft-2023 under a GPL license.

References

1. Abdelhafez, A., Alba, E., Luque, G.: A component-based study of energy consumption for sequential and parallel genetic algorithms. The Journal of Supercomputing **75**, 6194–6219 (2019)
2. Connolly Bree, D., Ó Cinnéide, M.: Energy efficiency of the visitor pattern: contrasting Java and C++ implementations. Empirical Software Engineering **28**(6), 145 (2023)
3. Corne, D., Lones, M.A.: Evolutionary Algorithms, p. 1–22. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-07153-4_27-1, http://dx.doi.org/10.1007/978-3-319-07153-4_27-1
4. Díaz-Álvarez, J., Castillo, P.A., Fernandez de Vega, F., Chávez, F., Alvarado, J.: Population size influence on the energy consumption of genetic programming. Measurement and Control **55**(1-2), 102–115 (2022)
5. Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. The Journal of Machine Learning Research **13**(1), 2171–2175 (2012)
6. Garg, K., Jindal, C., Kumar, S., Juneja, S.: Analyzing and rating greenness of nature-inspired algorithms. In: 6th International Conference on Innovative Computing and Communication (ICICC 2023) (2023)
7. Hidalgo, I., Fenández-de_Vega, F., Ceberio, J., Garnica, O., Velasco, J.M., Cortés, J.C., Villanueva, R., Díaz, J.: Sustainable artificial intelligence systems: An energy efficiency approach (2023)
8. Holló-Szabó, Á., Albert, I., Botzheim, J.: Statistical racing crossover based genetic algorithm for vehicle routing problem. In: 2021 IEEE 21st International Symposium on Computational Intelligence and Informatics (CINTI). pp. 000267–000272. IEEE (2021)
9. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The evolution of Lua. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. pp. 2–1 (2007)

10. Köhler, S., Herzog, B., Hönig, T., Wenzel, L., Plauth, M., Nolte, J., Polze, A., Schröder-Preikschat, W.: Pinpoint the Joules: Unifying runtime-support for energy measurements on heterogeneous systems. In: 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS). pp. 31–40 (2020). https://doi.org/10.1109/ROSS51935.2020.00009
11. Lannelongue, L., Aronson, H.E.G., Bateman, A., Birney, E., Caplan, T., Juckes, M., McEntyre, J., Morris, A.D., Reilly, G., Inouye, M.: Greener principles for environmentally sustainable computational science. Nature Computational Science **3**(6), 514–521 (2023)
12. Lima, L.G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., Fernandes, J.P.: Haskell in green land: Analyzing the energy behavior of a purely functional language. In: 2016 IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 517–528. IEEE (2016)
13. Merelo-Guervós, J.J., Blancas-Alvarez, I., Castillo, P.A., Romero, G., García-Sánchez, P., Rivas, V.M., García-Valdez, M., Hernández-Águila, A., Román, M.: Ranking the performance of compiled and interpreted languages in genetic algorithms. In: Proceedings of the International Conference on Evolutionary Computation Theory and Applications, Porto, Portugal. vol. 11, pp. 164–170 (2016)
14. Merelo-Guervós, J.J., Blancas-Álvarez, I., Castillo, P.A., Romero, G., García-Sánchez, P., Rivas, V.M., García-Valdez, M., Hernández-Águila, A., Román, M.: Ranking programming languages for evolutionary algorithm operations. In: Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I 20. pp. 689–704. Springer (2017)
15. Merelo-Guervós, J.J., García-Valdez, M., Castillo, P.A.: An analysis of energy consumption of JavaScript interpreters with evolutionary algorithm workloads. In: Fill, H., Mayo, F.J.D., van Sinderen, M., Maciaszek, L.A. (eds.) Proceedings of the 18th International Conference on Software Technologies, ICSOFT 2023, Rome, Italy, July 10-12, 2023. pp. 175–184. SCITEPRESS (2023). https://doi.org/10.5220/0012128100003538, https://doi.org/10.5220/0012128100003538
16. Merelo-Guervós, J.J., Romero, G., García-Arenas, M., Castillo, P.A., Mora, A.M., Jiménez-Laredo, J.L.: Implementation matters: Programming best practices for evolutionary algorithms. In: Cabestany, J., Rojas, I., Caparrós, G.J. (eds.) IWANN (2). Lecture Notes in Computer Science, vol. 6692, pp. 333–340. Springer (2011)
17. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Ranking programming languages by energy efficiency. Science of Computer Programming **205**, 102609 (2021). https://doi.org/https://doi.org/10.1016/j.scico.2021.102609, https://www.sciencedirect.com/science/article/pii/S0167642321000022
18. Pinto, G., Castor, F.: Energy efficiency: a new concern for application software developers. Communications of the ACM **60**(12), 68–75 (2017)
19. Theis, T.N., Wong, H.S.P.: The end of Moore's Law: A new beginning for information technology. Computing in science & engineering **19**(2), 41–50 (2017)
20. de Vega, F.F., Chávez, F., Díaz, J., García, J.A., Castillo, P.A., Merelo, J.J., Cotta, C.: A cross-platform assessment of energy consumption in evolutionary algorithms. In: Handl, J., Hart, E., Lewis, P.R., López-Ibáñez, M., Ochoa, G., Paechter, B. (eds.) Parallel Problem Solving from Nature – PPSN XIV. pp. 548–557. Springer International Publishing, Cham (2016)
21. Watson, R.A., Hornby, G.S., Pollack, J.B.: Modeling building-block interdependency. In: Parallel Problem Solving from Nature—PPSN V: 5th International Con-

ference Amsterdam, The Netherlands September 27–30, 1998 Proceedings 5. pp. 97–106. Springer (1998)