An Analysis of Energy Consumption of JavaScript Interpreters with Evolutionary Algorithm Workloads

Juan J. Merelo-Guervós[1][a], Mario García-Valdez[2][b] and Pedro A. Castillo[1][c]
[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*

Keywords: Green Computing, Metaheuristics, JavaScript, Energy-Aware Computing, Evolutionary Algorithms.

Abstract: What is known as energy-aware computing includes taking into account many different variables and parameters when designing an application, which makes it necessary to focus on a single one to obtain meaningful results. In this paper, we will look at the energy consumption of three different JavaScript interpreters: bun, node and deno; given their different conceptual designs, we should expect different energy budgets for running the (roughly) same workload, operations related to evolutionary algorithms (EA), a population-based stochastic optimization algorithm. In this paper we will first test different tools to measure per-process energy consumption in a precise way, trying to find the one that gives the most accurate estimation; after choosing the tool by performing different experiments on a workload similar to the one carried out by EA, we will focus on EA-specific functions and operators and measure how much energy they consume for different problem sizes. From this, we will try to draw a conclusion on which JavaScript interpreter should be used in this kind of workloads if energy (or related expenses) has a limited budget.

1 INTRODUCTION

From a language designed in the nineties for simple browser widgets and client-side validations (Goodman et al., 2007; Flanagan, 1998), JavaScript is nowadays the language most widely used by developers in their GitHub repositories (O'Grady, 2022), occupying this position since 2014 (O'Grady, 2014), mainly because it is almost exclusively the language needed for front-end programming (competing only with app development languages, such as Swift or Kotlin, or languages transpiled to JavaScript, such as Dart), while at the same time being strong for full-stack development, with solid support for the back end, including application servers, middleware, and database programming. Other popularity indices, such as TIOBE[1], that take into account other factors besides lines of code production, currently (2023) rank it as the seventh, although it was also the most popular language in 2014. It can be claimed, then, that it is among the most popular, if not the most popular language in current development.

Due to its popularity and the fact that it has a continuously evolving standard (ECMA, 1999), traditionally, there have been different virtual machines (or interpreters) to run its programs. During the first years, browsers were the only running platform available; however, the introduction of Node.js running on the V8 JavaScript Engine (Tilkov and Vinoski, 2010) gave it the popularity it has today; this popularity, in turn, provoked new interpreters to spawn like deno (Doglio,) (written in Rust) and bun (Tomar, 2022) programmed in the relatively unknown language Zig.

No wonder, then, that JavaScript is also a popular language for implementing metaheuristics, especially evolutionary algorithms (EA). EA (Eiben and Smith, 2015) are population-based stochastic optimization algorithms based on the representation of a problem as a (often binary) "chromosome", and *evolution* of population of these "chromosomes" by random change (via "genetic" operators, mutation and crossover) and survival of the fittest (evaluation of those chromosomes via a so-called "fitness" function, and selection and reproduction of those that achieve the best values). From the early implementations in the browser (Smith and Sugihara, 1996; González et al., 1999; Langdon, 2004), whole libraries (Ri-

[a] https://orcid.org/0000-0002-1385-9741
[b] https://orcid.org/0000-0002-2593-1114
[c] https://orcid.org/0000-0002-5258-0620
[1] https://www.tiobe.com/tiobe-index/

Merelo-GuervÃşs, J., GarcÃ■a-Valdez, M. and Castillo, P.
An Analysis of Energy Consumption of JavaScript Interpreters with Evolutionary Algorithm Workloads.
DOI: 10.5220/0012128100003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 175-184
ISBN: 978-989-758-665-1; ISSN: 2184-2833

vas et al., 2014), through complete implementations geared towards volunteer computing (Merelo et al., 2016). However, one of the criticisms leveraged towards these implementations is the (possible) lack of speed when compared to other compiled languages (mainly Java, very popular in metaheuristics implementations, or C++).

This is why, since implementation matters (Merelo-Guervós et al., 2011), choosing the right interpreter is going to have a significant impact on the performance of any workload; if you decide to choose JavaScript for any reason (such as seamless client/server integration, or be able to run your algorithm either on the browser or from the command line if desired) knowing which VM delivers the best performance is essential, either from the scientific, or software engineering points of view.

At the same time, with the advent of the concept of green computing (Kurp, 2008), it becomes increasingly important to measure not only the raw wall clock performance (which was the focus of papers such as (Merelo-Guervós et al., 2017)), but also to achieve a certain level of performance with a certain amount of energy consumption, or else to minimize the consumption needed to run a certain workload. This will be the main focus of this paper; since the core of the different JS virtual machines is different, and are created with languages with different focus (Rust is focused on memory safety (Noseda et al., 2022), Zig based on simplicity and performance (Kelley, 2019)), different energy consumption should be expected. Since all three languages can (roughly) run the same, unmodified source code, what we intend with this paper is to advise on which JS interpreter might give the lowest power consumption, the maximum performance, or both, so that EA practitioners can target it for their development.

The rest of the paper follows this plan: next we will present the state of the art; then we will describe the experimental setup in Section 3; results will be presented next in Section 4, and we will end with a discussion of results, conclusions and future lines of work.

2 STATE OF THE ART

The power efficiency of CPUs (computations per kilowatt-hour) has doubled roughly every year and a half from 1946 to 2009 (Koomey et al., 2011), this improvement has been mainly a by-product of Moore's law, the trend of chip manufacturers to decrease in half the size and distance between transistors every two years. Unfortunately, it is expected that physical limits of electronics will slow down this miniaturization in the near future. Nonetheless, energy efficiency is becoming the most important metric of performance and selling point in hardware development, and it is an important driver for current innovation. The challenge of building more power-efficient systems, can be addressed at the hardware and software levels. In the software level, developers focus their attention on the energy consumption of software, proposing optimizations for more energy-efficient algorithm implementations. Algorithm comparatives nowadays include power efficiency as a performance metric, these include encryption algorithms (Mota et al., 2017; Thakor et al., 2021), estimation models for machine learning applications (García-Martín et al., 2019) and genetic programming (GP) (Diaz Alvarez et al., 2018), and code refactoring (Ournani et al., 2021). Since metaheuristics are so extensively used in machine learning applications, its interest in research has grown in parallel to its number of applications. Many papers focus on analyzing how certain metaheuristics parameters have an impact on energy consumption. Díaz-Álvarez et al. (Díaz-Álvarez et al., 2022) studies how the populations size of EAs influences power consumption. In an earlier work, centered on genetic algorithms (GAs) (Fernández de Vega et al., 2020), power-consumption of battery-powered devices was measured for various parameter configurations including chromosome and population sizes. The experiments used the OneMax and Trap function benchmark problems, and they concluded that execution time and energy consumption do not linearly correlate and there is a connection between the GA parameters and power consumption. In GAs, the mutation operator appears to be a power-hungry component according to Abdelhafez et al. (Abdelhafez et al., 2019), in their paper they also report that in a distributed evaluation setting, the communication scheme has a grater impact. Fernández de Vega et al. (de Vega et al., 2016) experimented with different parameters for a GP algorithm and concluded that handheld devices and single-board computers (SBCs) required an order of magnitude less energy to run the same algorithm.

3 METHODOLOGY AND EXPERIMENTAL SETUP

There are many ways to measure the consumption of applications running in a computer; besides measuring directly from the power intake, those running as applications and tapping the computer sensors fall roughly into two fields: power monitors and energy

profilers (Cruz, 2021). Power monitors need additional hardware to measure the power drawn by the whole machine; besides being expensive, their setup is difficult, and it is complicated to measure precisely how much a specific process consumes.

On the other hand, energy profilers are programs that draw information from hardware sensors (Sinha and Chandrakasan, 2001), generally exposed through kernel calls or higher-level wrapper libraries, to pinpoint consumption by specific processes in a time period. Tools that give these measures, either with a graphic or a command line interface, have been available for some time already, and have become more popular lately. One of the mainstream processor architectures, Intel, includes an interface called RAPL, or Running Average Power Limit (Pandruvada, 2014). Essentially, it consists of a series of machine-specific registers (MSRs) that contain information on the wattage drawn by different parts of the architecture; the content of these registers will be processed (through the corresponding library) and consumed by different command line utilities. We will use these command line utilities since they produce an output that can be automatically processed and evaluated, which is what we are looking for in this paper[2].

Energy profiling, as measured by RAPL or other APIs, includes different *domains* (Khan et al., 2015), essentially the computing devices or peripherals requiring the reported amount of energy. DRAM or dynamic RAM, CORE, or GPU will report what happens on those specific devices, with a core being every one of the computing units within the central processing unit; other domains, like PKG or package, will report what happens in the "package", or CPU together with other devices in the chipset.

Considering that the available system has an AMD architecture, which is roughly compatible with the RAPL architecture, we will use two command line utilities, as they are the only ones available that can wrap the execution of a command and report on consumption for that specific command. These are open source tools that can be obtained directly from the corresponding repositories or by downloading and compiling their source code.

- pinpoint (Köhler et al., 2020) (available from https://github.com/osmhpi/pinpoint) is a tool that uses the RAPL interface, as well as the NVIDIA registers, to report the power consumed by these devices. In this paper, we will use it since it is the only one out of the three tools that can show the GPU consumption.
- perf[3] (Treibig et al., 2010) is a system tool that measures all kinds of performance events, including power consumption. It will be used mainly for the `pkg` domain; this is a domain that it is measured by all tools, but all of them would process device readings differently, so it will provide us with alternative estimations of the consumption.
- likwid-powermeter[4] measures the `CORE` as well as the `PKG` domain, which includes the former together with the so-called "uncore" components.

On the JavaScript side, we used three different interpreters:

- `bun` version 0.5.8
- `deno` version 1.32.1, which includes the v8 library version 11.2.214.9 and typescript 5.0.2
- `node.js` version 18.5.0

`bun` and `nodejs` are fully compatible, so they run exactly the same code. The code for `deno` needed a small modification: the path to the library had to be changed (since it does not use the `node_modules` to host installed modules), and it uses a different library for processing the command line arguments. Other than that, the business logic was exactly the same.

These were running in an Ubuntu version 20.04.1 with kernel version 5.15.0-69. The processor is an AMD Ryzen 9 3950X 16-Core. Since we will not be testing in a pure Intel architecture, the complete RAPL API is not going to be available; that is also why we will be experimenting with different tools so that we can have an adequate coverage of energy consumption for the commands we will be measuring.

A Perl script was created to perform the experiments; it launched the scripts and collected results by analyzing the standard output and putting it in a CSV format that would allow examination of the experiments.

The initial experiment consisted in a script that used the `saco-js` library to perform the union of "bags", sets that can hold several copies of the same item. 1024 sets were generated; these sets had 1024, 2048, 4096 elements. Then, a union of bags was performed on pairs of sets until there was only one left. This is similar to some operations performed by EAs, mainly related to merging populations. They do not involve floating point operations in any way.

[2]Systems based on the AMD architecture have a similar power profiling system called AMP with its corresponding command line tool. However, we found that it was not well documented, and excessively complicated for the purposes of this article. Although not as complete, AMD processors also include the aforementioned MSRs so RAPL-based utilities can run on them

[3]https://perf.wiki.kernel.org/index.php/Main_Page

[4]https://github.com/RRZE-HPC/likwid

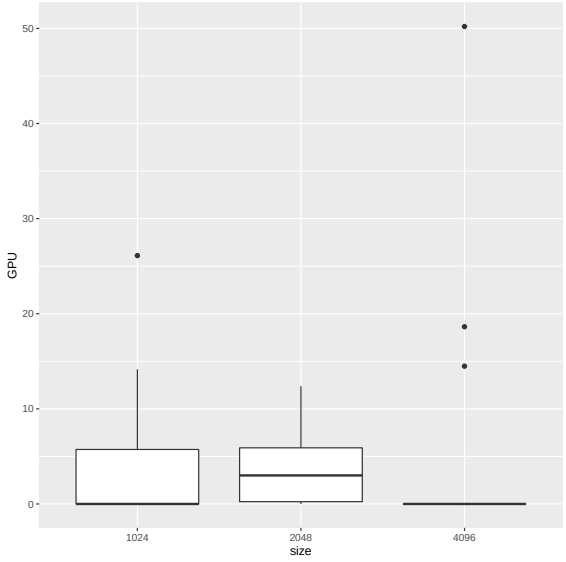

Figure 1: Boxplot of measurements of energy expenditures by the GPU in the sets problem. GPU energy consumption is measured in Joules.

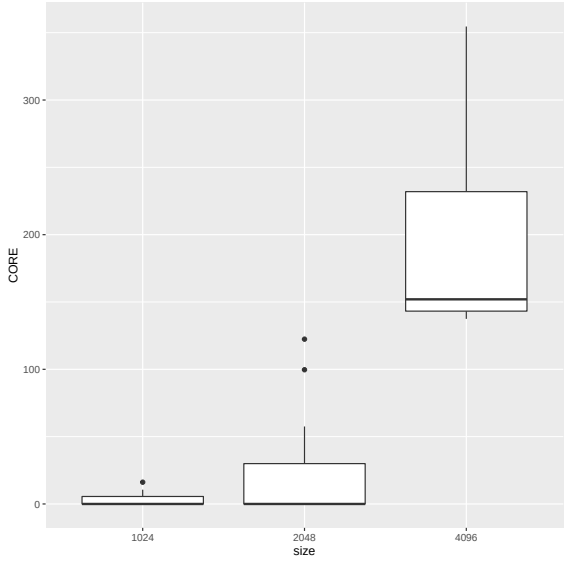

Figure 2: Boxplot of (differential) measurements of energy consumption as measured by the CORE and PKG (package) registers, both measured in Joules.

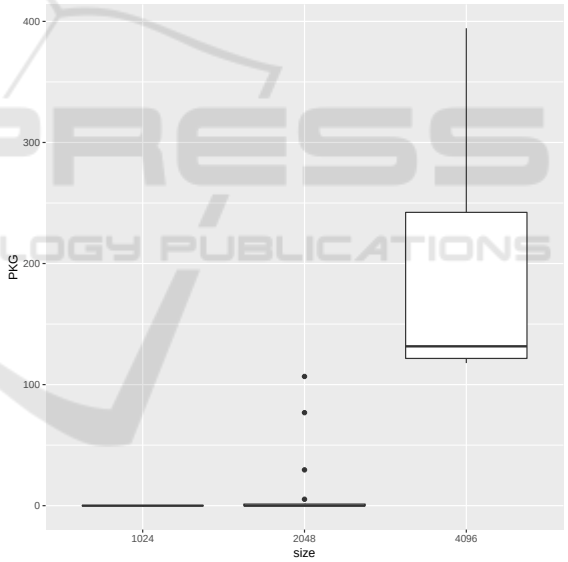

Figure 3: Boxplot of (differential) measurements of energy consumption as measured by the CORE and PKG (package) registers, both measured in Joules.

The scripts to launch every kind of tool were slightly different, mainly because the output needed to be processed in different ways (and different kinds of information extracted). Additionally, `pinpoint`, which is the only tool that does not need superuser privileges, sometimes returned 0 in energy measures. This was an error, and those runs were discarded.

Finally, the scripts performed an additional task: since it is not possible to disaggregate the readings for our program from the energy consumed by other processes running at the same time, what we did was to run every program 15 times, compute the average time, and then use the same tool to measure the energy consumption for the `sleep` program during the average amount of time. The energy readings shown are the result of subtracting this measurement from every one of the 15 other measurements taken, so that we can analyze the differential of energy that has been consumed by our programs; the result is clipped at 0, since negative energy differentials would make no sense.

Experimenting with this simple program will mainly allow us to calibrate the different tools in order to pick only one, if possible, as well as validate the program and iron out all possible errors in the program itself or the processing scripts.

One of the first observations we can draw from these initial experiments is whether measure how much energy the GPU spends it is interesting. Since `pinpoint` is the only tool able to measure this, we will use it. We show the results in Figure reffig:gpu. We see here that it is mostly independent of the set size, but most importantly, it is 0 in most cases, more than 50%. The rest must be essentially noise, amounting to a few Joules anyway. In order to decide if we can use this tool alone or complement it with other measures, we need to validate or enhance its measurements with others; that is why next, we will make some test measurements with the next tool, `likwid`.

Another tool that we have tested, `likwid`, can also measure the *package* energy consumption. In this initial exploration, we will see whether this mea-

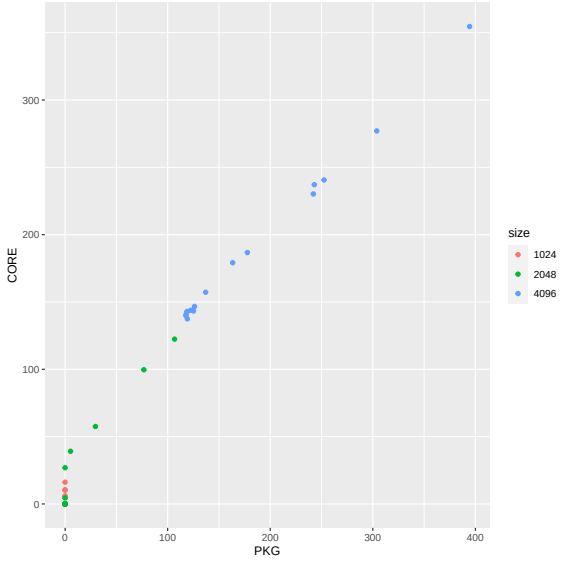

Figure 4: Relationship between CORE and PACKAGE consumption (x and y axis, respectively).

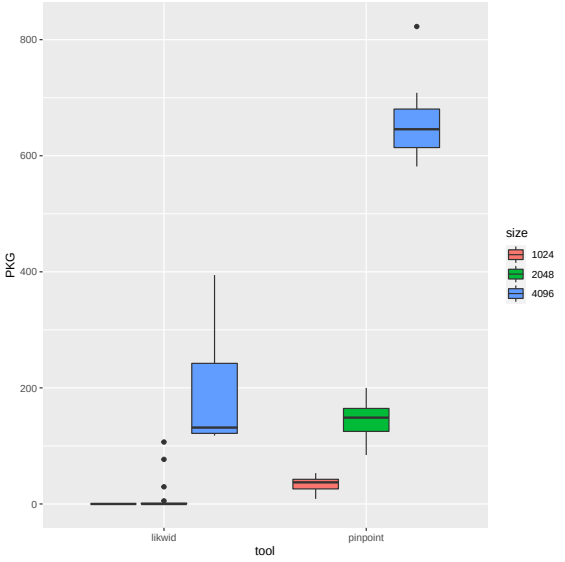

Figure 5: Relationship between PKG measurements taken by pinpoint and likwid.

surement is significant and to what extent it is related to the core measurements. Boxplots of energy consumption for different set size is shown in Figure 3 for the CORE (top) and PKG RAPL registers. It seems to show that differential energy consumption (once the baseline has been subtracted) is almost negligible in the CORE register unless the size is big enough (4096) in both cases, although the CORE register shows a certain amount of consumption for size = 2048. It is entirely unlikely that these kinds of measurements have a certain degree of uncertainty; however, apparently, `pinpoint`will directly estimate these runs as 0, and thus are skipped; however, this tool does measure a certain amount of consumption that cannot be so easily filtered.

Plotting the relationship between these two registers (see Figure 4), however, shows a linear relationship, so we do not really need to plot both. One of them will be enough, and since PKG seems to have the greatest variation, we will stick to that one.

Since several tools are available to measure PKG, we need to find out what kind of measurements they have, and if there is a correlation or even equality between them. We show how PKG energy consumption is registered by `pinpoint` and `likwid` in Figure 5. We have already seen in Figure 3 how the measurements registered by `likwid` have some strange behavior; this chart shows that while `pinpoint` shows a reasonable amount of consumption for all three sizes, `likwid` just registers zero, which is not reasonable in this case. This will lead us to discard the use of this tool in the upcoming experiments.

Unlike what happened with the previous tool, Figure 6 shows that there is a certain agreement between these two, except for the fact that `perf` seems to measure exceptionally low values in the lowest sizes. A Wilcox test shows significant differences for all three sizes; however, this might be due to a different amount of overhead or other unknown environmental factors.

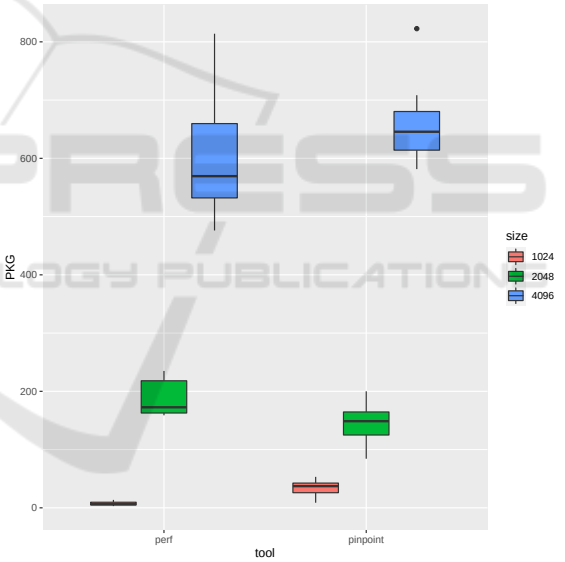

Figure 6: Relationship between PKG measurements taken by pinpoint and perf, with boxplots with values for the three set sizes.

Eventually, since `pinpoint` is able to measure all size ranges accurately, we will use it exclusively, as well as the quantity it measures, for comparison of different virtual machines. The initial results for this

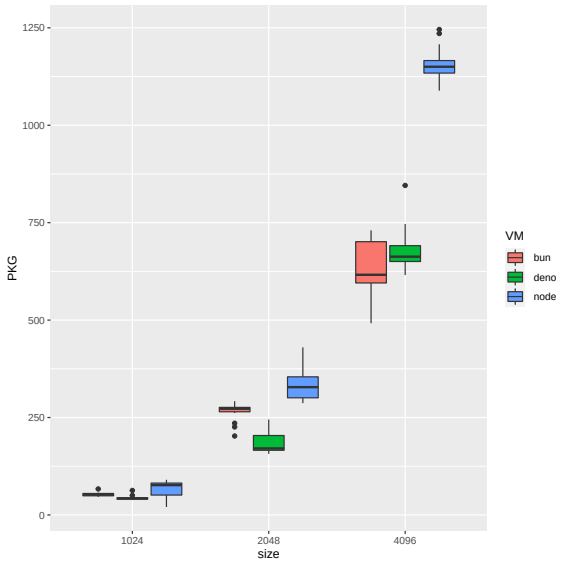

Figure 7: PKG measurements for the set problem and the three different virtual machines.

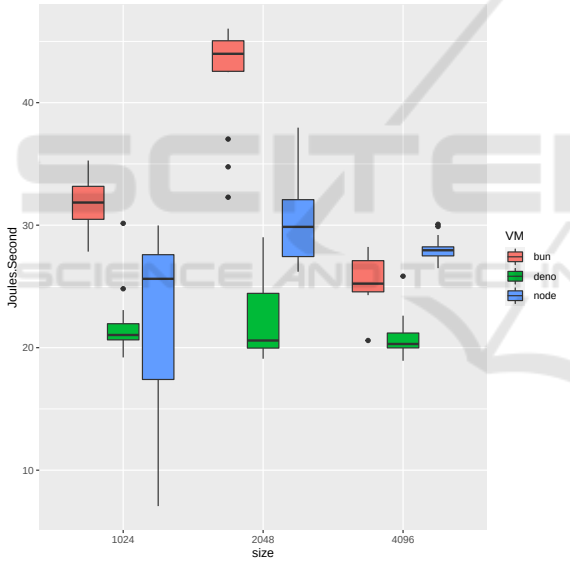

Figure 8: Boxplot of energy consumption per second, in Joules/s.

test drive are shown in Figure 7. This is due, in part, to the fact that they take a different amount of time, so it would be interesting to find out whether the energy density, or the consumption of energy per second, is similar. This is shown in Figure 8.

The interesting thing about this Figure is that the consumption per second varies with the VM used, as well as the size. Remarkably, deno keeps it approximately constant, while bun and node exhibit a variation with the size, and not in a systematic way. It is interesting also to note that while bun, in general, will spend less energy for each unit of work done than

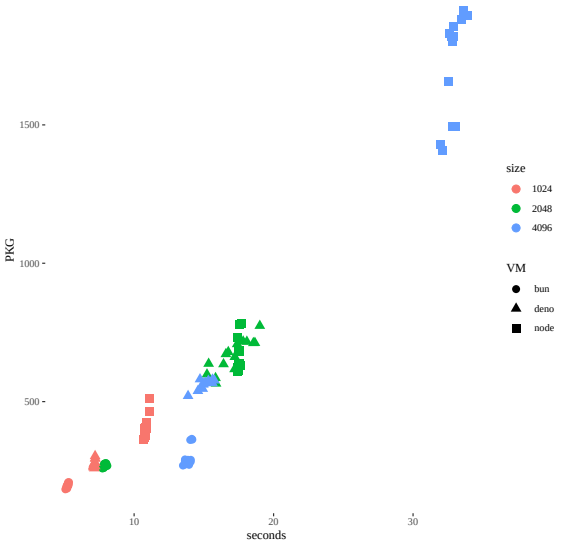

Figure 9: Boxplot of energy consumption vs. time taken for all three sizes and VMs.

node, it will do so at the expense of exercising the CPU package more strenuously, spending much more energy per second than the other two VMs.

These, however, are preliminary findings with a test problem; we will have to design experiments for actual operations used in EAs, which we will do next.

4 EXPERIMENTAL RESULTS

As was done in (Merelo et al., 2016), which was focused on wallclock performance, the experiments will be focused on the key operations performed by an EA: evaluation of fitness and ”genetic” operators like mutation and crossover. What we will do is, repeating the setup in the initial exploration, check the energy consumption for the processing of 40000 chromosomes, a number chosen to take a sizable amount of memory, but also on the ballpark of the usual number of operations in an EA benchmark, it is also small enough to not create garbage collection problems with the memory, something that was detected after the initial exploration. Experiments were repeated for the same chromosome size as before, 1024, 2048, and 4096, and for the three JS virtual machines used. Although the business logic is exactly the same for the experiments, the script run comes in two versions, one for deno and the other for bun/node, due to the different way they have of reading command-line arguments. This does not affect the overhead in any way. Code, as well as the data resulted from the experiments and analyzed in this paper, are released with a free license (along with this paper) from the reposi-

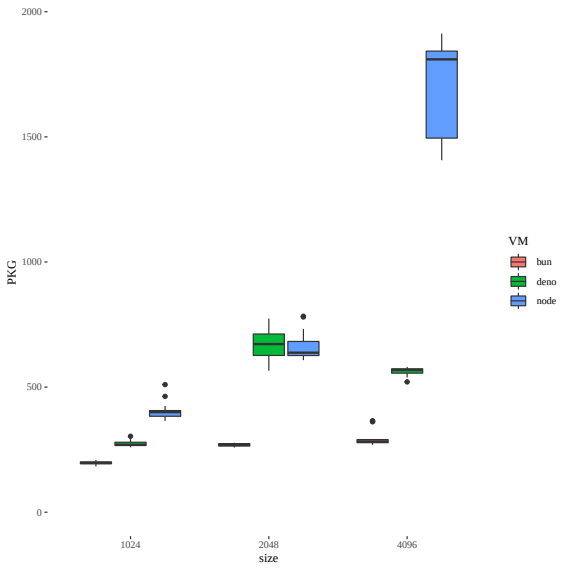

Figure 10: Boxplot of PKG measurements OneMax problem and the three different virtual machines.

tory https://github.com/JJ/energy-ga-icsoft-2023.

First, we will evaluate a typical fitness function, OneMax, which counts the number of ones in a binary (1s and 0s) string. This type of functions, which check the values of bits in a string and assign an integer value to it are typical of many papers focused on evaluating EAs, including parallel versions (Merelo Guervós and Valdez, 2018).

The results are shown in Figure 9. This already shows that time, as well as energy consumption, for node is higher; just check the separation of the squares representing individual experiments in that interpreter to the rest of the values for the same color; this separation increases with chromosome size. But this paper focuses on energy consumption, which we summarize next in Figure 10.

The figure shows the almost-flat growth of energy consumption for bun. How consumption grows for deno is weird, since it takes less energy when the chromosome is bigger (4096). Once again, node is the bigger energy guzzler, consuming up to 3 times more than deno on average, and more than 6 times as much as bun. We will see how this is reflected in monetary terms, taking into account that the cost in Spain today is around 0.2€/kWh. This cost, shown in table 1, reaches almost one-hundredth of a euro for the most ”expensive” VM, node; that gives you an idea of the kind of cost the algorithms have, and also how this cost decreases almost an order of magnitude if bun is used.

The crossover operation involves copy operations between strings, as well as creation of new strings. We will again generate 40K chromosomes and group

Table 1: Estimated cost of the OneMax runs for every VM and size, in €-cents.

size	VM	average	sd
1024	bun	0.0010921	0.0000419
1024	deno	0.0015286	0.0000730
1024	node	0.0022507	0.0002090
2048	bun	0.0014913	0.0000353
2048	deno	0.0037011	0.0003269
2048	node	0.0036966	0.0003216
4096	bun	0.0016519	0.0001929
4096	deno	0.0031257	0.0000933
4096	node	0.0094820	0.0010450

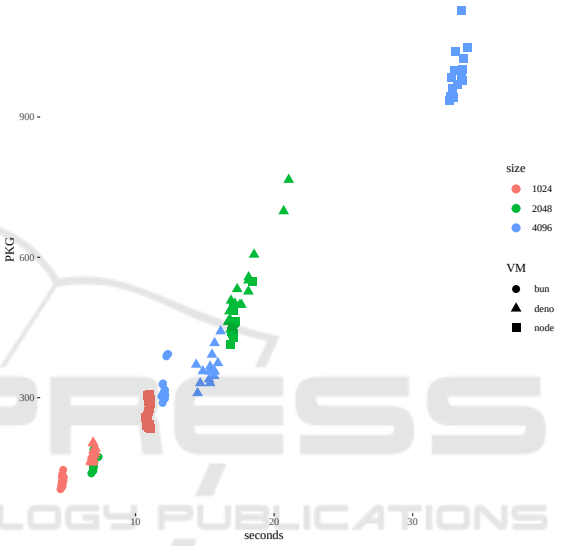

Figure 11: PKG consumption, in Joules, vs. time in seconds, for the crossover and the three different virtual machines.

them in pairs; these strings will be crossed by interchanging a random fragment from one to the other and back. The resulting pairs will be stored in an array, which is eventually printed. The result of every experiment is shown in an energy vs. wallclock time chart in Figure 11.

The scenario is remarkably similar to the one shown in Figure 9. In the two cases, bun achieves the top performance and lowest energy consumption, and node is the worst. Average energy consumption is shown as a boxplot in Figure 12.

Here we can see again the surprising fact that deno takes the same amount of energy, on average, as node for size 2048, in a similar case to what happened for OneMax (shown in Figure 10). The difference between the thriftiest, bun, and the heaviest consumer, node, is approximately three times, in this case, less than in the case of the OneMax fitness function.

Given that the results for the two EA-specific

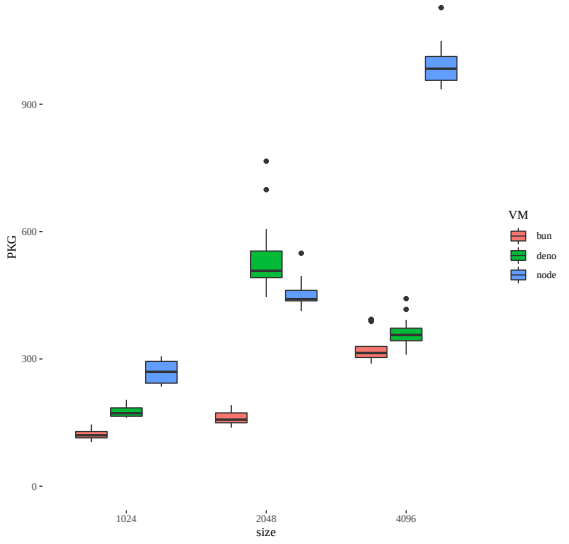

Figure 12: Boxplot of PKG measurements for the crossover operator and the three different virtual machines.

functions, as well as the test function, are quite conclusive, we should not expect anything different from the mutation function, so we will leave the experiments at these two, and proceed to the conclusions.

5 CONCLUSIONS

In this paper, we set out to study the energy efficiency of different JavaScript interpreters in EA workloads, by studying how short scripts that are extensively used in implementations of these algorithms work and how these energy expenses scale with the chromosome size. We have developed a methodology to accurately measure per-process consumption; we have also adopted a tool, `pinpoint`, that is able to give good estimations of sensor readings, discarding experiments where that estimation was not adequate; this tool has been calibrated by comparing its readings with other tools, which were also evaluated for the same purpose and eventually discarded. We have also adopted a benchmark-based approach, similar to the one used to measure performance, so that consumption for specific operations can be pinpointed, discarding noise produced by an implementation of a complete algorithm; that is, a whole program includes different operations, applied in different proportion, that might cancel each other. Testing short code paths, as proposed by (Hähnel et al., 2012) makes easier to understand their individual contribution to the overall consumption of the algorithm, and eventually optimize their specific code, or the number of times they are applied in the algorithm.

During the exploratory data analysis, we have established that, in Linux machines, `pinpoint` can be profitably used to measure per-process energy consumption, as long as these measurements are repeated and the process themselves include short snippets of business logic; this tool should be preferred over others that are either less accurate or simply take into account different aspects of energy consumption.

The main point of this paper, however, was to check which JavaScript interpreter should be used if our objective is to consume the minimum amount of energy; the experiments have reliably confirmed bun to be that tool. Not only it consumes less for all the range of chromosome sizes used; it also takes less time and can run applications written for Node (mostly) unmodified; its consumption also scales better with problem size. This might be due to design considerations, but also to the fact that the language used to write it, Zig, emphasizes compile-time safety and manual memory allocation by default, and avoids hidden control flow. The only inconvenience of this interpreter is that it has not reached version 1.0 yet, being currently in version 0.5.9; this might prevent any company or organization from using it in production environments.

If that is an issue, deno might be a good alternative. Except in a specific case, it is going to be faster and consume less energy than node, even more so when memory requirements are high. According to our initial exploration, it will also consume less energy *per second*, thus for workloads that take roughly the same time, it will be a better candidate than node. As an inconvenience, it needs minor modifications to run, at least if you need core or other kind of external libraries; its core library modules are different to those used in node/bun, although that need not be a disadvantage per se.

The previous two points imply that, energy-wise, there are no good reasons to use node.js for running EAs. Except if the business logic uses specific, early-adoption, or some features that, for some reason, does not work with bun yet, we would advise anyone to keep using bun for this kind of workloads.

As we have indicated in the experimental session, the wide advantage that bun has over the other interpreters does not leave much room for adopting different benchmarks that could make that ranking vary; at any rate, these experiments have shown how much faster and energy-saving bun is (from 1/3 to 1/6 the energy consumed by node), but it would be interesting to know what happens to this gap under different operations like selection or different kind of mutation. At the same time, even if EAs are mostly GPU-free, there are some fitness functions that operate on float-

ing point numbers and thus would need to use the GPU; how interpreters work in this area could be an interesting future line of work. This, along with testing different versions of the interpreters as they are published, will be the subject of future research.

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).

REFERENCES

Abdelhafez, A., Alba, E., and Luque, G. (2019). A component-based study of energy consumption for sequential and parallel genetic algorithms. *The Journal of Supercomputing*, 75:6194–6219.

Cruz, L. (2021). Tools to measure software energy consumption from your computer. https://luiscruz.github.io/2021/07/20/measuring-energy.html.

de Vega, F. F., Chávez, F., Díaz, J., García, J. A., Castillo, P. A., Merelo, J. J., and Cotta, C. (2016). A cross-platform assessment of energy consumption in evolutionary algorithms. In Handl, J., Hart, E., Lewis, P. R., López-Ibáñez, M., Ochoa, G., and Paechter, B., editors, *Parallel Problem Solving from Nature – PPSN XIV*, pages 548–557, Cham. Springer International Publishing.

Díaz-Álvarez, J., Castillo, P. A., Fernandez de Vega, F., Chávez, F., and Alvarado, J. (2022). Population size influence on the energy consumption of genetic programming. *Measurement and Control*, 55(1-2):102–115.

Diaz Alvarez, J., Castillo Martínez, P. A., Rodríguez Díaz, F. J., Fernández de Vega, F., et al. (2018). A fuzzy rule-based system to predict energy consumption of genetic programming algorithms.

Doglio, F. Introducing Deno.

ECMA (1999). 262: ECMAScript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*.

Eiben, A. E. and Smith, J. E. (2015). *Introduction to evolutionary computing*, chapter What is an evolutionary algorithm?, pages 25–48. Springer.

Fernández de Vega, F., Díaz, J., García, J. Á., Chávez, F., and Alvarado, J. (2020). Looking for energy efficient genetic algorithms. In Idoumghar, L., Legrand, P., Liefooghe, A., Lutton, E., Monmarché, N., and Schoenauer, M., editors, *Artificial Evolution*, pages 96–109, Cham. Springer International Publishing.

Flanagan, D. (1998). *JavaScript*. O'Reilly.

García-Martín, E., Rodrigues, C. F., Riley, G., and Grahn, H. (2019). Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88.

González, J., Merelo-Guervós, J.-J., Castillo, P. A., Rivas, V., Romero, G., and Prieto, A. (1999). Optimized web newspaper layout using simulated annealing. In Mira, J. and Sánchez-Andrés, J. V., editors, *Engineering Applications of Bio-Inspired Artificial Neural Networks, International Work-Conference on Artificial and Natural Neural Networks, IWANN '99, Alicante, Spain, June 2-4, 1999, Proceedings, Volume II*, volume 1607 of *Lecture Notes in Computer Science*, pages 759–768. Springer.

Goodman, D., Morrison, M., and Eich, B. (2007). *JavaScript® Bible*. John Wiley & Sons, Inc. New York, NY, USA.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17.

Kelley, A. (2019). Introduction to the Zig programming language. https://andrewkelley.me/post/intro-to-zig.html.

Khan, K. N., Nybäck, F., Ou, Z., Nurminen, J. K., Niemi, T., Eulisse, G., Elmer, P., and Abdurachmanov, D. (2015). Energy profiling using IgProf. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1115–1118. IEEE.

Koomey, J. G., Berard, S., Sanchez, M., and Wong, H. (2011). Web extra appendix: implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):S1–S30.

Kurp, P. (2008). Green computing. *Communications of the ACM*, 51(10):11–13.

Köhler, S., Herzog, B., Hönig, T., Wenzel, L., Plauth, M., Nolte, J., Polze, A., and Schröder-Preikschat, W. (2020). 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)*, pages 31–40.

Langdon, W. (2004). Global Distributed Evolution of L-Systems Fractals. In *Genetic Programming: 7th European Conference, EuroGP 2004, Coimbra, Portugal, April 5-7, 2004, Proceedings*, volume 7, pages 349–358. Springer.

Merelo, J. J., Castillo, P. A., Blancas, I., Romero, G., García-Sánchez, P., Fernández-Ares, A., Rivas, V. M., and Valdez, M. G. (2016). Benchmarking languages for evolutionary algorithms. In Squillero, G. and Burelli, P., editors, *Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part II*, volume 9598 of *Lecture Notes in Computer Science*, pages 27–41. Springer.

Merelo, J.-J., García-Valdez, M., Castillo, P. A., García-Sánchez, P., de las Cuevas, P., and Rico, N. (2016). NodIO, a JavaScript framework for volunteer-based evolutionary algorithms : first results. *ArXiv e-prints*.

Merelo-Guervós, J. J., Blancas-Alvarez, I., Castillo, P. A., Romero, G., García-Sánchez, P., Rivas, V. M., Valdez, M. G., Hernández-Águila, A., and Román, M. (2017). Ranking programming languages for evolutionary algorithm operations. In Squillero, G. and Sim, K., editors, *Applications of Evolutionary Computation - 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I*, volume 10199 of *Lecture Notes in Computer Science*, pages 689–704.

Merelo-Guervós, J.-J., Romero, G., García-Arenas, M., Castillo, P. A., Mora, A.-M., and Jiménez-Laredo, J.-L. (2011). Implementation matters: Programming best practices for evolutionary algorithms. In Cabestany, J., Rojas, I., and Caparrós, G. J., editors, *IWANN (2)*, volume 6692 of *Lecture Notes in Computer Science*, pages 333–340. Springer.

Merelo Guervós, J. J. and Valdez, J. M. G. (2018). Performance improvements of evolutionary algorithms in Perl 6. In Aguirre, H. E. and Takadama, K., editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 1371–1378. ACM.

Mota, A. V., Azam, S., Shanmugam, B., Yeo, K. C., and Kannoorpatti, K. (2017). Comparative analysis of different techniques of encryption for secured data transmission. In *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, pages 231–237. IEEE.

Noseda, M., Frei, F., Rüst, A., and Künzli, S. (2022). Rust for secure IoT applications: why C is getting rusty. In *Embedded World Conference 2022, Nuremberg, 21-23 June 2022*. WEKA.

O'Grady, S. (2014). The RedMonk Programming Language Rankings: January 2014. Tecosystems blog.

O'Grady, S. (2022). The RedMonk Programming Language Rankings: October 2022. Tecosystems blog.

Ournani, Z., Rouvoy, R., Rust, P., and Penhoat, J. (2021). Tales from the code# 1: The effective impact of code refactorings on software energy consumption. In *ICSOFT 2021-16th International Conference on Software Technologies*.

Pandruvada, S. (2014). Running Average Power Limit – RAPL. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl. Accessed on March 28, 2023.

Rivas, V. M., Merelo-Guervós, J. J., Romero-López, G., Arenas-García, M., and Mora, A. M. (2014). An object-oriented library in JavaScript to build modular and flexible cross-platform evolutionary algorithms. In Esparcia-Alcázar, A. I. and Mora, A. M., editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 853–862. Springer Berlin Heidelberg.

Sinha, A. and Chandrakasan, A. P. (2001). Jouletrack: A web based tool for software energy profiling. In *Proceedings of the 38th annual Design Automation Conference*, pages 220–225.

Smith, J. and Sugihara, K. (1996). GA toolkit on the Web. In *Proceedings of the First Online Workshop on Soft Computing*, page 12.

Thakor, V. A., Razzaque, M. A., and Khandaker, M. R. (2021). Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities. *IEEE Access*, 9:28177–28193.

Tilkov, S. and Vinoski, S. (2010). Node.js: Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80–83.

Tomar, D. (2022). Bun JS : A brand-new, lightning-quick JavaScript runtime. *Medium*. https://devangtomar.medium.com/bun-a-brand-new-lightning-quick-javascript-runtime-e42119a306ca.

Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th international conference on parallel processing workshops*, pages 207–216. IEEE.