Minimizing evolutionary algorithms energy consumption in the low-level language Zig

Juan J. Merelo-Guervós[1] [0000−0002−1385−9741]

Department of Computer Engineering, Automatics and Robotics, University of Granada, Granada, Spain

Abstract. Managing energy resources in scientific computing implies awareness of a wide range of software engineering techniques that, when applied, can minimize the energy footprint of experiments. In the case of evolutionary computation, we are talking about a specific workload that includes the generation of chromosomes and operations that change parts of them or access and operate on them to obtain a fitness value. In a low-level language such as Zig, we will show how different choices will affect the energy consumption of an experiment.

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

One of the concerns in modern evolutionary computing is reducing the amount of energy spent in experiments, trying to make nature-inspired computing more nature-friendly [6]. This involves developing a methodology to measure energy spent, as well as identifying the EC operations that consume the most energy. In [4] we settled on a language and OS-independent set of tools, but also followed [1] in choosing the set of operations under measure: mutation, crossover and a simple fitness evaluation, ONEMAX.

In [4] the main factor under study was the different interpreters used in a high-level language, JavaScript. In the case of low-level languages like `zig`, a language that emphasizes safety and maintainability [2], there is a single compiler, but there are several choices to be made, even if the defaults should provide enough performance and energy efficiency. Yet, in general, developers, and even more so scientific ones, are generally unaware of the energy impact of their algorithm implementations [5], not to mention techniques available for their reduction [3].

In this paper we will work on a generic evolutionary algorithm workload and see what the impact of different choices will have on its energy consumption. With this, we will try to find some best practices that will help practitioners implement evolutionary algorithms in `zig` or other low-level languages.

The experiment setup will match the one used in [4], using the same tools for energy profiling (`pinpoint`) as well as Perl scripts to run the experiments and process the results. 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, with zig version 0.11.0, released by August 3, 2023, which is the last stable one at the time of writing this paper. The Perl scripts generate CSV files that are then processed and plotted using R embedded in the source code of this paper. All code, data and source for this paper are available at https://github.com/JJ/energy-ga-icsoft2023 under a free license.

There are several units whose consumption can be measured using pinpoint via the RAPL interface; since the use of GPU is negligible in these examples, only memory and CPUs will be measured. Together, they are called the *package* (alongside with caches and memory controllers); this is usually represented by the acronym PKG.

We will be examining choices in three different areas

– By default, zig adds debug information to the resulting binary, without performing any kind of optimization. We will test the impact of using the ReleaseFast option when building binaries.
– The first version used strings for representing chromosomes. We will test arrays of Boolean values instead, which is a primitive type too.
– Unlike other languages, zig provides different memory allocators, which can be chosen by the developer. By default, a page allocator is used, but there is the possibility of using a fixed buffer size allocator.

We will first generate 40000 chromosomes of size 512, 1024 and 2048, and measure the energy consumption and running time of this operation; every combination is run 15 times. Not all combinations of the three techniques above could be tested, we show the results in Figure 1, along with the baseline that was compiled using default settings, character strings and page allocator. The other choices tested are: Built with ReleaseFast option (tagged "ReleaseFast"), built in the same way and using a Fixed Buffer Allocator ("ReleaseFastFBA"), built with default options, using Boolean arrays as well as the Fixed Buffer Allocator ("Boolean")[1].

This Figure 1 shows the dramatic change in energy consumption (as well as performance) that can be obtained just by changing compiling and programming options. Using a fixed buffer alongside the ReleaseFast option implies a 95% reduction in the energy consumed[2]. Even with the default compilation options, applying changes in the allocator and the data structure used reduces by 50% energy used. The impact on running time, although not the focus of this paper, is also significant.

We will now run an experiment that, after generating the 40K chromosomes, will perform crossover + mutation + onemax operations on chromosomes of size 512, 1024 and 2048 [3]. In this case, the combination of fixed buffer allocator plus ReleaseFast has been skipped, since beyond the initial generation of

[1] This combination proved difficult in practice, crashing the program for size 2048; it did not work with the ReleaseFast option either

[2] We should take into account that generating chromosomes uses allocation heavily, so this is the kind of operation that would be the most impacted

[3] As indicated above, we could not make the Boolean bitstring along with fixed buffer allocation work for the biggest size

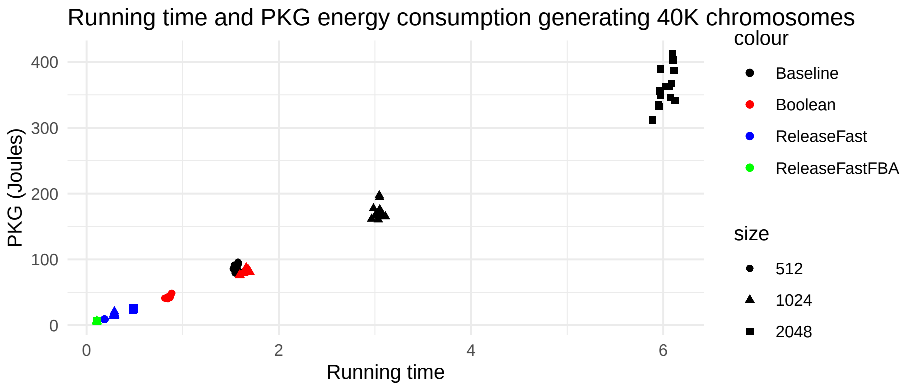

Fig. 1. Average running time and PKG energy consumption generating 40K chromosomes for the different techniques used (represented with different colors); dot shape represents the chromosome size.

chromosomes, there is barely any allocation happening; in fact, just two temporary bitstrings during the crossover operation. Comparisons for the other two combinations plus baseline are shown in Figure 2.

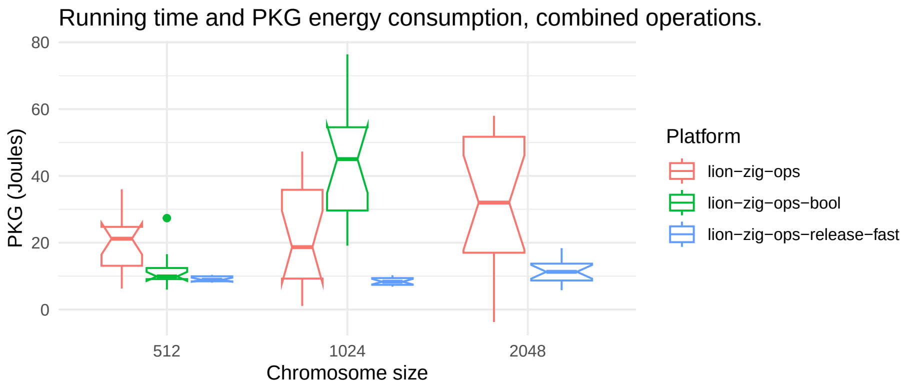

Fig. 2. Boxplot of PKG energy consumption processing 40K chromosomes via crossover, mutation and ONEMAX for different combinations of optimization techniques in Zig

This case is remarkably different to the one shown in Figure 1. For starters, the energy consumption is very low, one order of magnitude lower; regular genetic operations do not spend a lot of energy indeed, with most consumption focused on fitness functions. Even so, the fast compilation uses half the energy of the other combinations. Remarkably, using a different data structure does not always imply a better energy profile; it does so only for the smaller size, 512 bits (which is anyway closer to usual chromosome sizes).

In this paper we have used different techniques that potentially could reduce the amount of energy spent by a genetic algorithm implemented using the low-level language zig: testing different data structures, allocation policies and compilation options. The most dramatic reduction is achieved with the *fast* compilation policy, but additional improvements can be obtained by using adequate data structures and allocation policies. By using best practices in this area, zig implementations can indeed be considered the *greener* ones among other high-level or Java Virtual Machine based languages. It remains as future work, however, how applying similar techniques when available will impact the energy consumption of other languages.

Acknowledgements and data availability

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 for helpful tips. Source and data 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. Friesen, A.: Designing programming languages for writing maintainable software (2023), https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1625&context=honorstheses
3. Longo, M., Rodriguez, A.V., Mateos Diaz, C.M., Zunino Suarez, A.O.: Reducing energy usage in resource-intensive java-based scientific applications via micro-benchmark based code refactorings (2019), http://hdl.handle.net/11336/121006
4. 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
5. Pang, C., Hindle, A., Adams, B., Hassan, A.E.: What do programmers know about software energy consumption? IEEE Software **33**(3), 83–89 (2016). https://doi.org/10.1109/MS.2015.83
6. Fernández de Vega, F., Díaz, J., García, J.Á., Chávez, F., Alvarado, J.: Looking for energy efficient genetic algorithms. In: Idoumghar, L., Legrand, P., Liefooghe, A., Lutton, E., Monmarché, N., Schoenauer, M. (eds.) Artificial Evolution. pp. 96–109. Springer International Publishing, Cham (2020)