Best practices for energy-thrifty evolutionary algorithms in the low-level language zig

Juan J. Merelo-Guervós[1][0000−0002−1385−9741], Antonio M. Mora[2][0000−0003−1603−9105], and Mario García-Valdez[3][0000−0002−2593−1114]

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

Abstract. The most fruitful way of making evolutionary algorithms spend the least amount of energy is to consider all possible programming techniques and platform choices that could, theoretically, affect performance, and carry out experiments using EA workloads in different platforms, eventually choosing those techniques that yield the minimum amount of energy expenses. These techniques include a choice of different data structures, as well as affecting compilation in such a way that energy footprint is reduced; they have to be replicated in different computing platforms because these expenditures may be affected by all the layers of the operating system and runtime framework used. In this paper we will experiment with different data structures and code refactoring techniques in the low-level language zig, trying to design rules of thumb that will help developers create *green* evolutionary algorithms. We will include two different hardware platforms, looking for the one that spends the least energy.

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

A straightforward methodology for measuring energy consumption of EA implementations with the objective of reducing it would consider a baseline implementation and then change the code at different levels, measuring the resulting energy expenses. In [2], the main factor under study was the different interpreters used in a high-level language, JavaScript. In this paper, we will focus on different techniques applied to the low-level language zig, a language that emphasizes safety and maintainability [1], and that has as a motto "no implicit memory allocation," unlike other languages like C or C++, that will allocate memory without the user noticing. This strict memory management has several implications in terms of programming, but also gives the programmer more control over how and when memory is allocated and deallocated.

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, hopefully extensible to other low-level languages (which could include C and C++, but also Rust or Go).

The experiment setup will match the one used in [2], using the same tools for energy profiling (pinpoint) as well as Perl scripts to run the experiments and process the results. This way we can easily compare the results, but also use an established and proved methodology.

The experiments for this paper will be carried out in two different platforms:

- A Linux machine `5.15.0-94-generic #104 20.04.1-Ubuntu SMP` using AMD Ryzen 9 3950X 16-Core Processor.
- An M1 MacBook Air with 16GB of RAM and macOS Ventura 13.2.1.

We use zig version 0.11.0, released by August 3, 2023, which is the last stable one at the time of writing this paper. The pinpoint tool has no versions, but we have used one compiled from source and commit hash `1578db0`. Outputs of pinpoint are processed by Perl scripts that generate CSV files that are then processed and plotted using R embedded in the source code of this paper. This paper's code, data, and source 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 caches and memory controllers); this is usually represented by the acronym PKG. In the case of the Mac, this measurement is divided into three parts: the "E" (efficiency) and "P" (performance) CPUs, and the memory. Again, these user-initiated processes run on the "P" CPUs, so that will be the one we will be measuring.

By default, all programs will be generated using the `.ReleaseFast` compile option, that optimizes performance, but also energy consumption. We will use the `page_allocator` that allocates memory in the heap. This is an efficient allocator, but also the default choice.

We will be examining options in different areas

- Several data structures used in the implementation of EAs will be checked for: the default string, arrays of Boolean values and bit sets.
- The default crossover operator used an allocator to create temporary copies of chromosomes. Several implementations will be tested.
- Unlike other languages, zig provides different memory allocators, which the developer can choose. By default, a page allocator is used, but there is the possibility of using a fixed buffer size allocator.
- We will check the behavior of these on two different platforms, Linux (with AMD CPU) and MacOS (with M1).

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.

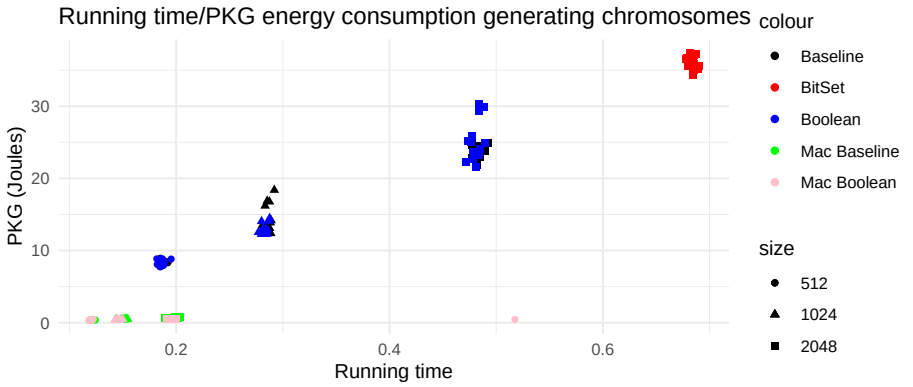

Fig. 1. Average running time and PKG energy consumption generating 40K chromosomes for the different parametrizations used (represented with different colors); dot shape represents the chromosome size.

Figure 1 represents energy consumption, as well as time taken, for the different configurations. The first thing to notice, in the upper right corner, is that using a bitset in zig will not represent any energy saving, with a big difference in time as well as energy consumption; this is why it has not even been tested for the Mac. Second feature that stands out is that consumption as well as time is very different for the Mac, which is an entry-level laptop computer, and the AMD desktop computer. This is probably expected, but the fact is that the MacBook Air, which is a generation behind current commercial offerings, takes 25% of the time and 10% of the energy to do the same amount of work. While increase in energy consumption with chromosome size is quite steep in the AMD architecture used by the desktop computer, that is not the case for the Mac, with a very small increase from the smallest to the largest size. Another observation is that, except for using bitsets as data structures, there does not seem to be a significant difference between using either strings or Boolean arrays in neither architecture, at least for this baseline operation. However, generating chromosomes is done essentially once, so it is not the most significant operation in EAs.

Thus, 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 using the strings and Boolean arrays in the two architectures used above.

What Figure 2 shows is a remarkable difference in energy consumption in the Mac platform, which goes against the big difference shown in the generation of chromosomes. This probably reveals a problem in the implementation of some feature used by the evolutionary algorithm. Digging into the code, we found that the main issue was the need to use allocation within the crossover operator. A refactoring, which eliminated this need to use allocators, was in order, yielding

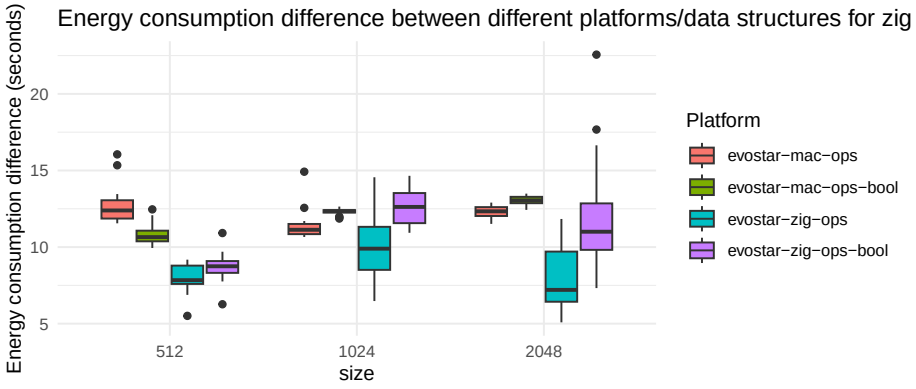

Fig. 2. Boxplot of PKG energy consumption processing 40K chromosomes via crossover, mutation and ONEMAX for different combinations of optimization techniques and platforms in Zig

the results shown in Figure 3. The energy consumption is now more in line with the generation of the chromosomes, and the difference between the different compilation policies is more in line with the previous results.

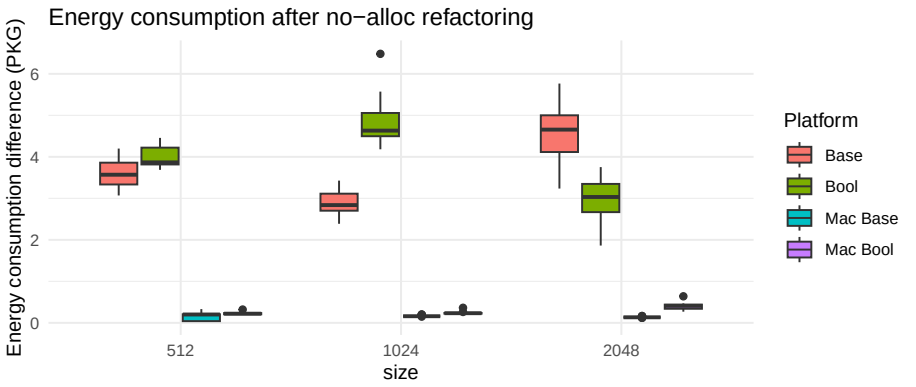

Fig. 3. Boxplot of PKG energy consumption processing 40K chromosomes via crossover, mutation and ONEMAX after crossover has been refactored

Figure 3 shows the energy consumption after refactoring. Both platforms show a considerable improvement, but it is more dramatic in the case of the Mac platform. We can anyway observe that, although there is a certain difference between both data structures, string and Boolean array, that difference does not hold across platforms and sizes. Strings seem to best the other option in more ocassions, but when the size is the biggest, it might be at a certain disadvantage.

It is probably the case that the amount of memory used by any of them is the same, and the random-access structure is also similar in performance and energy consumption (and unlike bitsets, which probably used more memory, since they internally used structs). On the other hand, eliminating allocation from the crossover operator results in energy savings across the board, although they are much more dramatic in the Mac platform, consuming energy for 40k evaluations that is a fraction of a Joule, and almost two orders of magnitude less of what the desktop computer consumes.

This leads us to the conclusion, carried over from other papers, that rather than taking assumptions on the behavior of implementations based on first principles or knowledge of the computing platform, we need to always create energy profiles of the implementations, and measure for different data structures and across refactoring of the evolutionary algorithm code.

It seems clear, anyway, that platforms that emphasize energy savings like the M1 chip used by the MacBookAir will use much less energy, dramatically so in some cases, than desktop solutions based on AMD. This does not extend to the operating system implementation itself: operations that depend on it, such as memory allocation, will have a different impact on the energy profile, which might be one of the reasons they should be minimized whenever possible. Fortunately, zig is a language and toolchain with very strict control over memory allocation, allowing us to be very conscious over where it could be eliminated, as we have done in this case.

As a final conclusion, implementing an EA in zig and running in on a Mac may result in an improvement of several orders of magnitude in consumption of energy over using high-level platforms (like Python or Javascript) and desktop machines. If we strive for greener computing, we should really consider them for our experiments. Although zig cannot be considered mainstream right now, its performance and capability should make it a very interesting option for the future.

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. Source and data available from https://github.com/JJ/energy-ga-icsoft-2023 under a GPL license.

References

1. Friesen, A.: Designing programming languages for writing maintainable software (2023), https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1625&context=honorstheses
2. 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