Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption

Kyriakos Georgiou
University of Bristol
Bristol, UK
Kyriakos.Georgiou@bristol.ac.uk

Craig Blackmore
University of Bristol
Bristol, UK
Craig.Blackmore@bristol.ac.uk

Samuel Xavier-de-Souza
Universidade Federal do Rio Grande do Norte
Natal, Brazil
samuel@dca.ufrn.br

Kerstin Eder
University of Bristol
Bristol, UK
Kerstin.Eder@bristol.ac.uk

Abstract

This paper presents the interesting observation that by performing fewer of the optimizations available in a standard compiler optimization level such as -O2, while preserving their original ordering, significant savings can be achieved in both execution time and energy consumption. This observation has been validated on two embedded processors, namely the ARM Cortex-M0 and the ARM Cortex-M3, using two different versions of the LLVM compilation framework; v3.8 and v5.0. Experimental evaluation with 71 embedded benchmarks demonstrated performance gains for at least half of the benchmarks for both processors. An average execution time reduction of 2.4% and 5.3% was achieved across all the benchmarks for the Cortex-M0 and Cortex-M3 processors, respectively, with execution time improvements ranging from 1% up to 90% over the -O2. The savings that can be achieved are in the same range as what can be achieved by the state-of-the-art compilation approaches that use iterative compilation or machine learning to select flags or to determine phase orderings that result in more efficient code. In contrast to these time consuming and expensive to apply techniques, our approach only needs to test a limited number of optimization configurations, less than 64, to obtain similar or even better savings. Furthermore, our approach can support multi-criteria optimization as it targets execution time, energy consumption and code size at the same time.

CCS Concepts • **Software and its engineering → Compilers**; • **General and reference →** *Measurement*; *Metrics*; *Performance*;

Keywords Autotuning, compiler optimizations, embedded systems, execution time, energy consumption, phase-ordering

ACM Reference Format:
Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de-Souza, and Kerstin Eder. 2018. Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption. In *SCOPES '18: 20th International Workshop on Software and Compilers for Embedded Systems, May 28–30, 2018, Sankt Goar, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3207719.3207727

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany

ACM ISBN 978-1-4503-5780-7/18/05…$15.00
https://doi.org/10.1145/3207719.3207727

1 Introduction

Compilers were introduced to abstract away the ever-increasing complexity of hardware and improve software development productivity. At the same time, compiler developers face a hard challenge: producing optimized code. A modern compiler supports a large number of architectures and programming languages and it is used for a vast diversity of applications. Thus, tuning the compiler optimizations to perform well across all possible applications is impractical. The task is even harder as compilers need to adapt to rapid advancements in hardware and programming languages.

Modern compilers adopted two main practices to mitigate the problem and find a good balance between the effort needed to develop compilers and their effectiveness in optimizing code. The first approach is the splitting of the compilation process into distinct phases. Modern compilers such as those based on the LLVM compilation framework [17], allow for a common optimizer that can be used by any architecture and programming language. This is made possible by the use of an Intermediate Representation (IR) language on which optimizations are applied. Then a front-end framework is provided to allow programming languages to be translated into the IR, and a back-end framework exists that allows the IR to be translated into specific instruction set architectures (ISA). Therefore, to take advantage of the common optimizer one only needs to create a new front-end for a programming language and a new back-end for an architecture.

The second practice is the use of standard optimization levels, typically -O0, -O1, -O2, -O3 and -Os. Most modern compilers have a large number of transformations exposed to software developers via compiler flags; for example, the LLVM's optimizer has 56 documented transformations [18]. There are two major challenges a software developer faces while using compilers. First, to select the right set of transformations, and second to order the chosen transformations in a meaningful way, also called the compiler phase-ordering problem. The common objective is to achieve the best resource usage based on the application's requirements. To address this, each standard optimization level offers a predefined sequence of optimizations, which are proven to perform well based on a number of micro-benchmarks and a range of architectures. For example, for the LLVM compilation framework, starting from the -O0 level, which has no optimizations enabled, and moving to -O3, each level offers more aggressive optimizations with the main focus being performance, while -Os is focused on optimizing code size. Code size is critical for embedded applications with a

limited amount of memory available. Furthermore, the optimization sequences defined for each level encapsulate the accumulated empirical knowledge of compiler engineers over the years. For example, some optimizations depend on other code transformations being applied first, and some optimizations offer more opportunities for other optimizations. Note that a code transformation is not necessarily an optimization, but instead, it can facilitate an IR structure which enables the application of other optimizations. Thus, a code transformation does not always lead to better performance.

Although standard optimization levels are a good starting point, they are far from optimal in many cases, depending on the application and architecture used. An optimization configuration is a sequence of ordered flags. Due to the huge number of possible flag combinations and their possible orderings, it is impractical to explore the whole optimization-configuration space. Thus, finding optimal optimization configurations is still an open challenge. To tackle this issue, iterative compilation and machine-learning techniques have been used to find good optimization sequences by exploiting only a fraction of the optimization space [7]. Techniques involving iterative compilation are expensive since typically a large amount of optimization configurations, in the order of hundreds to thousands, need to be exercised before reaching any performance gains over standard optimization levels. On the other hand, machine learning approaches require a large training phase and are hardly portable across compilers and architectures.

This paper takes a different approach. Instead of trying to explore a fraction of the whole optimization space, we are focusing on exploiting the existing optimization levels. For example, using the optimization flags included in the -O2 optimization level as a starting point, a new optimization configuration is generated each time by removing the last transformation flag of the current optimization configuration. In this way, each new configuration is a subsequence of the -O2 configuration, that preserves the ordering of flags in the original optimization level. Thus, each new optimization configuration stops the optimization earlier than the previously generated configuration did. This approach aims to preserve the empirical knowledge built into the ordering of flags for the standard optimization levels. The advantages of using this technique are:

- The architecture and the compiler are treated as a black box, and thus, this technique is easy to port across different compilers or versions of the same compiler, and different architectures. To demonstrate this we applied our approach to two embedded architectures (Arm Cortex-M0 and Cortex-M3) and two versions of the LLVM compilation framework (v3.8 and v5.0);
- An expensive training phase similar to the ones needed by the machine learning approaches is not required;
- The empirical knowledge built into the existing optimization levels by the compiler engineers is being preserved;
- In contrast to machine-learning approaches and random iterative compilation [9], which permit reordering transformation passes, our technique retains the original order of the transformation passes. Reordering can break the compilation or create a malfunctioning executable;
- In contrast to the majority of machine-learning approaches, which are often opaque, our technique provides valuable insights to the software engineer on how each optimization flag affects the resource of interest;

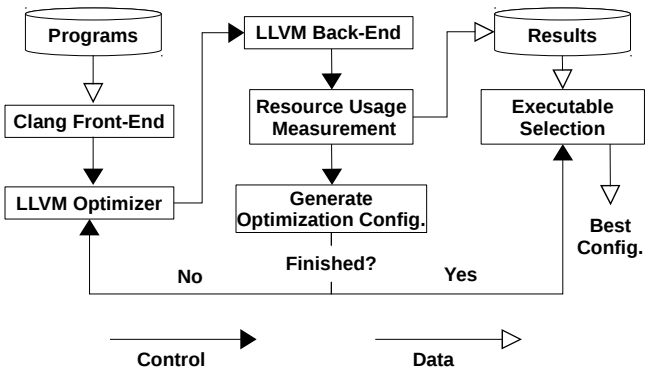


Figure 1. Compilation and evaluation process.

- Because energy consumption, execution time and code size of each optimization configuration are being monitored during compilation, multi-criteria optimizations are possible without needing to train a new model for each resource.

Our experimental evaluation demonstrates an average of 2.4% and 5.3% execution time improvement for the Cortex-M0 and Cortex-M3 processors, respectively. Similar savings were achieved for energy consumption. These results are in the range of what existing complicated machine learning or time consuming iterative compilation approaches can offer on the same embedded processors [8, 22].

The rest of the paper is organized as follows. Section 2 gives an overview of the compilation and analysis methodology used. Our experimental evaluation methodology, benchmarks and results are presented and discussed in section 3. Section 4 critically reviews previous work related to ours. Finally, Section 5 concludes the paper and outlines opportunities for future work.

2 Compilation and Analysis

As the primary focus of this work is deeply embedded systems, we demonstrate the portability of our technique across different architectures by exploring two of the most popular embedded processors: the Arm Cortex-M0 [2] and the Arm Cortex-M3 [3]. Although the two architectures belong to the same family, they have significant differences in terms of performance and power consumption characteristics [4]. The technique treats an architecture as a black box as no resource models are required e.g. energy-consumption or execution-time models. Instead, execution time and energy consumption physical measurements are used to assess the effectiveness of a new optimization configuration on a program.

For demonstrating the portability of the technique across different compiler versions, the analysis for the Cortex-M0 processor was performed using the LLVM compilation framework v3.8., and for the Cortex-M3 using the LLVM compilation framework v5.0. The technique treats the compiler as a black box since it only uses the compilation framework to exercise the different optimization-configuration scenarios, extracted from a predefined optimization level, on a particular program. In contrast, machine-learning-based techniques typically require a heavy training phase for each new compiler version or when a new optimization flag is introduced [6, 8].

Figure 1 demonstrates the process used to evaluate the effectiveness of the different optimization configurations explored. Each configuration is a set of ordered flags used to drive the analysis and transformation passes by the LLVM optimizer. An analysis pass can identify properties and expose optimization opportunities that can later be used by transformation passes to perform optimizations. A standard optimization level (-O1, -O2, -O3, -Os, -Oz) can be selected as the starting point. Each optimization level represents a list of optimization flags which have a predefined order. Their order influences the order in which the transformation/optimization and analysis passes will be applied to the code under compilation. A new flag configuration is obtained by excluding the last transformation flag from the current list of flags. Then the new optimization configuration is being applied to the unoptimized intermediate representation (IR) of the program, obtained from the Clang front-end. Note that the program's unoptimized IR only needs to be generated once by the Clang front-end; it can then be used throughout the exploration process thus saving compilation time. The optimized IR is then passed to the LLVM back-end and linker to generate the executable for the architecture under consideration. Note that both the back-end and linker are always called using the optimization level selected for exploration; in our case -O2. The executable's energy consumption, execution time and code size are measured and stored. The exploration process finishes when the current list of transformation flags is empty. This is equivalent to optimization level -O0, where no optimizations are applied by the optimizer. Then, depending on the resource requirements, the best flag configuration is selected.

There are two kinds of pass dependencies for the LLVM optimizer; explicit and implicit dependencies. An explicit dependency exists when a transformation pass requires an other analysis pass to execute first. In this case, the optimizer will automatically schedule the analysis pass if only the transformation pass was requested by the user. An implicit dependency exists when a transformation or analysis pass is designed to work after another transformation instead of an analysis pass. In this case, the optimizer will not schedule the pass automatically, instead the user must manually add the passes in the correct order to be executed either using the *opt* tool or the *pass manager*. The *pass manager* is the LLVM built-in mechanism for scheduling passes and handling their dependencies. If a pass is requested but its dependencies have not been requested in the correct order, then the specified pass will be automatically skipped by the optimizer. For the predefined optimization levels, the implicit dependencies are predefined in the *pass manager*.

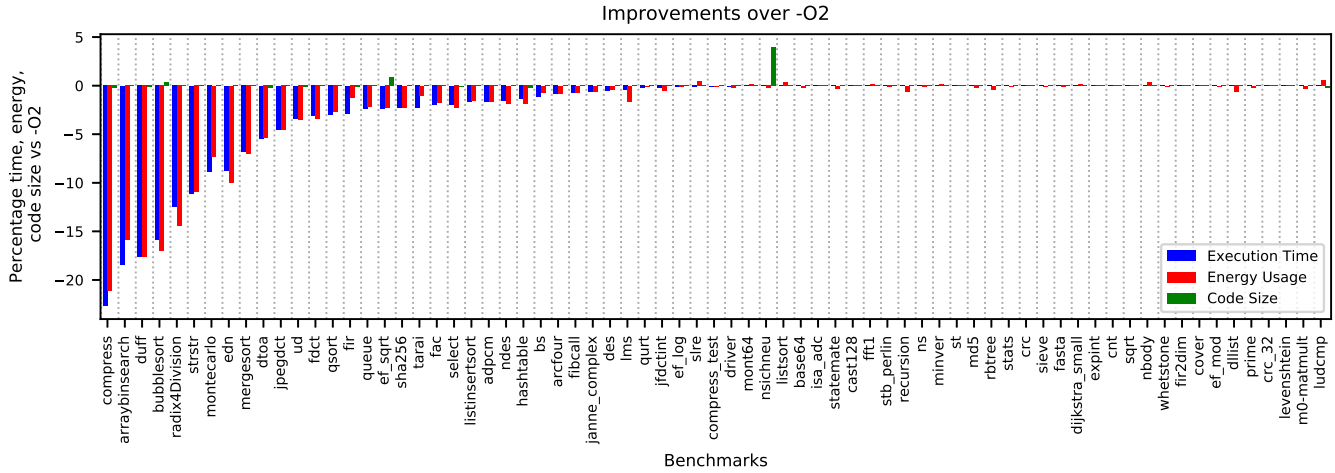
To extract the list of transformation and analysis passes, their ordering, and their dependencies for a predefined level of optimization, we use the argument "-debug-pass=Structure" with the *opt* tool (the LLVM optimizer). This information is passed to our flag-selection process, which, to extract a new configuration, simply eliminates the last optimization flag applied. This ensures that all the implicit dependencies for the remaining passes in the new configuration are still in place. Thus, the knowledge built into the predefined optimization levels about effective pass orderings is preserved in the newly generated optimization configurations. What we are actually questioning is whether the pass scheduling in the predefined-optimization levels is a good choice. In other words, can stopping the optimizations at an earlier point yield more optimal code for a specific program and architecture?

The BEEBS benchmark suite [21] was used for evaluation. BEEBS is designed for assessing the energy consumption of embedded processors. The resource usage estimation process retrieves the execution time, energy consumption and code size for each executable generated. The code size can be retrieved by examining the size of the executable. The execution time and energy consumption is being measured using the MAGEEC board [16] together with the pyenergy [20] firmware and host-side software. The BEEBS benchmark suite utilizes this energy measurement framework and allows for triggering the begin and the end of the execution of a benchmark. Thus, energy measurements are reported only during a benchmark's execution. Energy consumption, execution time and average power dissipation are reported back to the host. The MAGEEC board supports a sampling rate of up to six million samples per second. A calibration process was needed prior to measurement to determine the number of times a benchmark should be executed in a loop while measuring to obtain an adequate number of measurements. This ensured the collection of reliable energy values for each benchmark. Finally, the BEEBS benchmark suite has a built-in self-test mechanism that flags up when a generated executable is invalid, i.e. it does not provide the expected results. Standard optimization levels shipped with each new version of a compiler are typically heavily tested to ensure the production of functionally correct executables. In our case, using optimization configurations that are subsequences of the standard optimization levels increases the chance of generating valid executables. In fact, all the executables we tested passed the BEEBS validation.

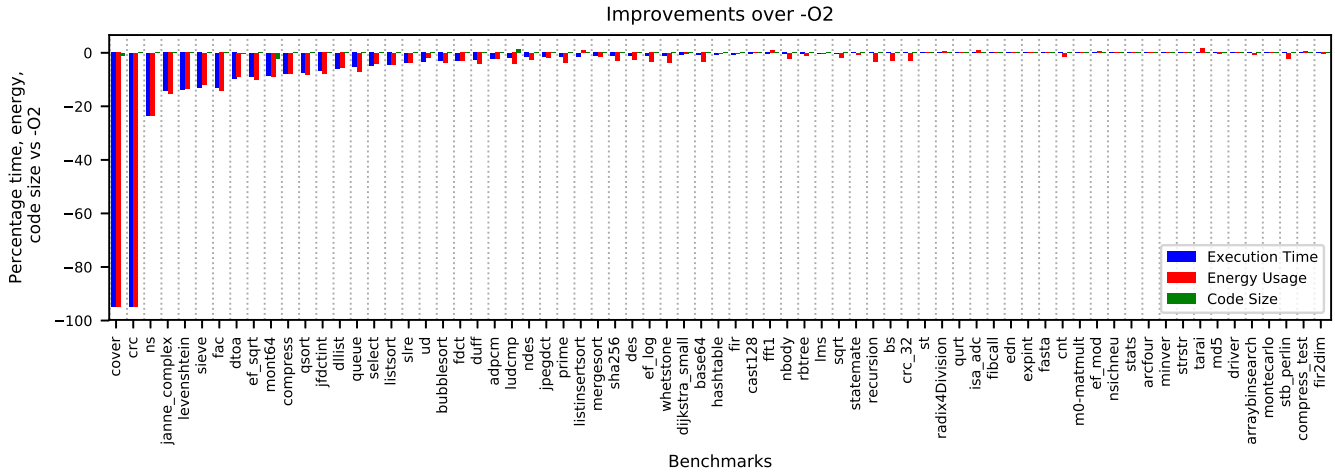
3 Results and Discussion

For the evaluation of our approach, the same 71 benchmarks from the BEEBS [21] benchmark suite were used for both the Cortex-M0 and the Cortex-M3 processors. Two benchmarks were left out because they did not fit into the memory of the Cortex-M0 development board. For each benchmark, Figure 2 (Figure 2a for the Cortex-M0 and the LLVM v3.8 and Figure 2b for the Cortex-M3 and the LLVM v5.0) demonstrates the biggest performance gains achieved by the proposed technique compared to the standard optimization level under investigation, -O2. In other words, this figure represents the resource usage results obtained by using the optimization configuration, among the configurations exercised by our technique, that achieves the best performance gains compared to -O2 for each benchmark. A negative percentage represents an improvement on a resource, e.g. a result of -20% for execution time represents a 20% reduction in the execution time obtained by the selected optimization configuration when compared to the execution time retrieved by -O2. The energy-consumption and code-size improvements are also given for the selected configurations. If two optimization configurations have the same performance gains, then energy consumption improvement is used as a second criterion and code size improvement as a third criterion to select the best optimization configuration. The selection criteria can be modified according to the resource requirements for a specific application. Moreover, a function can be introduced to further formalize the selection process when complex multi-objective optimization is required.

For the Cortex-M0 processor, we observed an average reduction in execution time of 2.5%, with 29 out of the 71 benchmarks seeing execution time improvements over -O2 ranging from around 1% to



(a) Results for the Cortex-M0 processor and the LLVM v3.8 compilation framework.



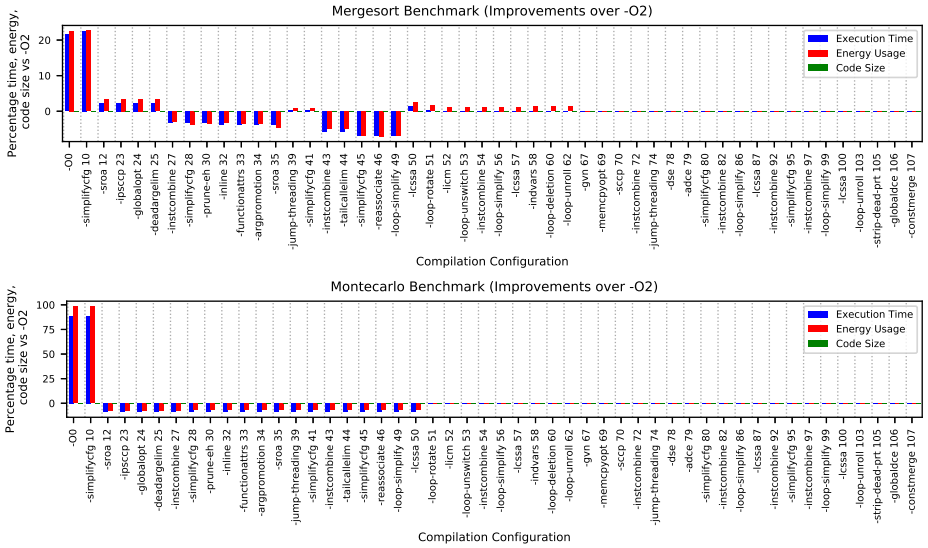
(b) Results for the Cortex-M3 processor and the LLVM v5.0 compilation framework.

Figure 2. Best achieved execution-time improvements over the standard optimization level -O2. For the best execution-time optimization configuration, energy consumption and code size improvements are also given. A negative percentage represents a reduction of resource usage compared to -O2.

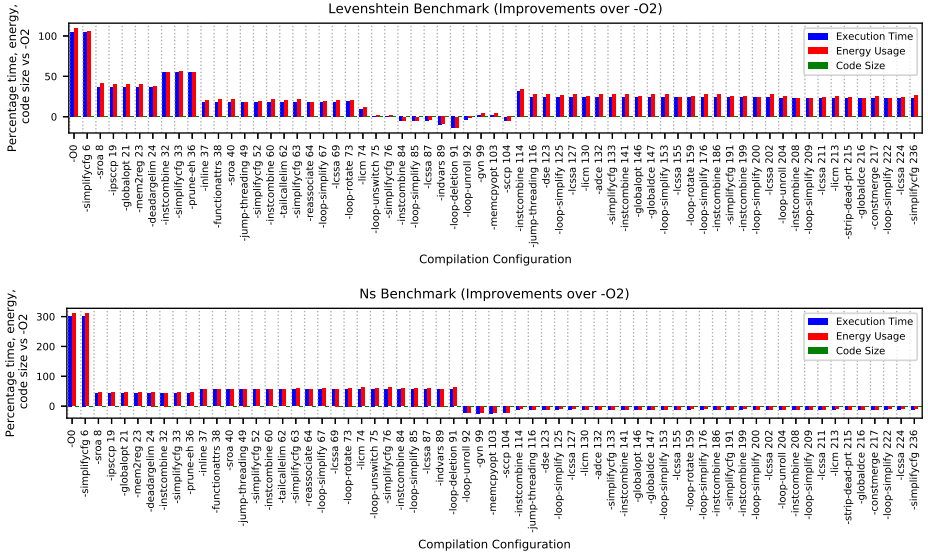
around 23%. For the Cortex-M3 processor, we observed an average reduction in execution time of 5.3%, with 38 out of the 71 benchmarks seeing execution time improvements over -O2 ranging from around 1% to around 90%. The energy consumption improvements were always closely related to the execution time improvements for both of the processors. This is expected due to the predictable nature of these deeply embedded processors. In contrast, there were no significant fluctuations in the code size between different optimization configurations. We anticipate that, if the -Os or -Oz optimization levels, which both aim to achieve smaller code size, had been used as a starting point for our exploration, then more variation would have been observed for code size.

As it can be seen from Figures 2a and 2b, our optimization strategy performed significantly different for the two processors per benchmark. This can be caused by the different performance and power consumption characteristics of the two processors and/or the use of different compiler versions in each case. Furthermore, the technique performed better on the Cortex-M3 with the LLVM v5.0 compilation framework. This could be due to the compilation framework improvements from version 3.8 to version 5.0. Another possible reason might be that the -O2 optimization level for LLVM v5.0 includes more optimization flags than the LLVM v.3.8. The more flags in an optimization level, the more optimization configurations will be generated and exercised by our exploitation technique, and thus, more opportunities for execution-time, energy-consumption and code-size savings can be exposed.

Figures 3a and 3b demonstrate the effect of each optimization configuration, exercised by our exploitation technique, on the three resources (execution time, energy consumption and code size), for two of the benchmarks for the Cortex-M0 and Cortex-M3 processors, respectively. Similar figures were obtained for all the 71 benchmarks and for both of the processors. Similarly to Figure 2, a negative percentage represents an improvement on the resource compared to the one achieved by -O2. The horizontal axis of the



(a) Compilation profiles for two of the benchmarks, using the Cortex-M0 processor and the LLVM v3.8 compilation framework.



(b) Compilation profiles for two of the benchmarks, using the Cortex-M3 processor and the LLVM v5.0 compilation framework

Figure 3. For each optimization configuration tested by the proposed technique the execution-time, energy-consumption and code-size improvements over -O2 are given. A negative percentage represents a reduction of resource usage compared to -O2. Each element of the horizontal axis has the name of the last flag applied and the total number of flags used. The configurations are incremental subsequences of the -O2, starting from -O0 and adding optimization flags till reaching the complete -O2 set of flags.

figures shows the flag at which compilation stopped together with the total number of flags included up to that point. This represents an optimization configuration that is a subsequence of the -O2. For example, the best optimization configuration for all three resources for the *Levenstein* benchmark (see top part of Figure 3b) is achieved when the compilation stops at flag number 91, *-loop-deletion*. This means that the optimization configuration includes the first 91 flags of the -O2 configuration with their original ordering preserved. The optimization configurations include both transformation and analysis passes.

The number of optimization configurations exercised in each case depends on the number of transformation flags included in the -O2 level of the version of the LLVM optimizer used. Note that we are only considering the documented transformation passes [18]. For example, 50 and 64 different configurations are being tested in the case of the Cortex-M0 processor with the LLVM compilation framework v3.8, and the case of Cortex-M3 with the LLVM framework v5.0, respectively. Many of the transformation passes are repeated multiple times in a standard optimization level, but because of their different ordering, they have a different effect. Thus, we consider each repetition as an opportunity to create a new optimization configuration. Furthermore, note that more transformation passes exist in the LLVM optimizer, but typically, these are passes that have implicit dependencies on the documented passes. The methodology of creating a new optimization configuration explained in Section 2 ensures the preservation of all the implicit dependencies for each configuration. This is part of preserving the empirical knowledge of good interactions between transformations built into the predefined optimization levels and reusing it in the new configurations generated.

Typically, optimization approaches based on iterative compilation are extremely time consuming [6], since thousands of iterations are needed to reach levels of resource savings similar to the ones achieved by our approach. In our case the maximum number of iterations we had to apply were the 64 iterations for the Cortex-M3 processor. This makes our simple and inexpensive approach an attractive alternative, before moving to the more expensive approaches, such as iterative-compilation-based and machine-learning-based compilation techniques [5, 7].

By manually observing the compilation profiles obtained for all the benchmarks, similar to the ones demonstrated in Figure 3, no common behavior patterns were detected, except that typically there is a significant improvement on the execution time and the energy consumption at the third optimization configuration, i.e. the *sroa 12* and the *sroa 8* configurations shown in Figure 3 for the Cortex-M0 and Cortex-M3 processors, respectively. Future work will use clustering to see if programs can be grouped together based on their compilation profiles. This can be useful to identify optimization sequences that perform well for a particular type of program. Furthermore, the retrieved optimization profiles can also give valuable insights to compiler engineers and software developers on the effect of each optimization flag on a specific program and architecture. It is beyond the scope of this work to investigate these effects.

4 Related Work

Iterative compilation has been proved an effective technique for tackling both the problems of choosing the right set of transformations and for ordering them to maximize their effectiveness [6]. The technique is typically used to iterate over different sets of optimizations with the aim of satisfying an objective function. Usually, each iteration involves some feedback, such as profiling information, to evaluate the effectiveness of the tested configuration. In random iterative compilation [9], random optimization sequences are generated, ranging from hundreds to thousands, and then used to optimize a program. Random iterative compilation has been proved to provide significant performance gains over standard optimization levels. Thus, it has become a standard baseline metric for evaluating the effectiveness of machine-guided compilation approaches [6, 8, 12], where the goal is to achieve better performance gains with less exploration time. Due to the huge number of possible flag combinations and their possible orderings, it is impossible to explore a large fraction of the optimization space. To mitigate this problem, machine learning is used to drive iterative compilation [1, 10, 19].

Based on either static code features [12] or profiling information [10], such as performance counters, machine learning algorithms try to predict the best set of flags to apply to satisfy the objective function with as few iterations as possible. The techniques have proven to be effective in optimizing the resource usage, mainly execution-time, of programs on a specific architecture but generally suffer from a number of drawbacks. Typically, these techniques require a large training phase [19] to create their predictive models. Furthermore, they are hardly portable across different compilers or versions of the same compiler and different architectures. Even if a single flag is introduced to the set of a compiler's existing flags the whole training phase has to be repeated. Moreover, extracting some of the metrics that these techniques depend on, such as static code features, might require a significant amount of engineering.

A recent work that is focused on mitigating the phase-ordering problem, [6], divided the -O3 standard optimization flags of the LLVM compilation framework v3.8, into five subgroups using clustering. Then they used iterative compilation and machine learning techniques to select optimization configurations by reordering the subgroups. The approach demonstrated average performance speedup of 1.31. An interesting observation is that 79% of the -O3 optimization flags were part of a single subgroup with a fixed ordering that is similar to that used in the -O3 configuration. This suggests that the ordering of flags in a predefined optimization level is a good starting point for further performance gains. Our results actually confirm this hypothesis for the processors under consideration.

Embedded applications typically have to meet strict timing, energy-consumption, and code-size constraints [11, 13]. Handwritten optimized code is a complex task and requires extensive knowledge of architectures. Therefore, utilizing the compilers optimizations to achieve optimal resource usage is critical.

In an attempt to find better optimization configurations than the ones offered by the standard optimization levels, the authors in [8] applied inductive logic programming (ILP) to predict compiler flags that minimize the execution time of software running on embedded systems. This was done by using ILP to learn logical rules that relate effective compiler flags to specific program features. For their

experimental evaluation they used the GCC compiler, [23], and the Arm Cortex-M3 architecture; the same architecture used by this paper. Their method was evaluated on 60 benchmarks selected from the BEEBS benchmark suite; the same used in this work. They were able to achieve an average reduction in execution time of 8%, with about half of the benchmarks seeing performance improvements. The main drawback of their approach was the large training phase of their predictive model. For each benchmark, they needed to create and test 1000 optimization configurations. This resulted in about a week of training time. Furthermore, for their approach to be transferred to a new architecture, compiler or compiler version, or even to add a new optimization flag, the whole training phase has to be repeated from scratch. The same applies for applying their approach to resources other than execution time, such as energy consumption or code size. In contrast, our approach, for the same architecture and more benchmarks of the same benchmark suite, was able to achieve similar savings in execution time (average 5.3%) by only testing 65 optimization configurations for each program. At the same time, our approach does not suffer from the portability issues faced by their technique.

In [22], the authors used fractional factorial design (FFD) to explore the large optimization space (2^{82} possible combinations for the GCC compiler used) and determine the effects of optimizations and optimization combinations. The resources under investigation were execution time and energy consumption. They tested their approach on five different embedded platforms including the Cortex-M0 and Cortex-M3, which are also used in this work. For their results to be statistically significant, they needed to exercise 2048 optimization configurations for each benchmark. Although they claimed that FFD was able to find optimization configurations that perform better than the standard optimization levels, they demonstrated this only on a couple of benchmarks. Again, this approach suffers from the same portability issues as [8].

In our work, to maximize the accuracy of our results, hardware measurements were used for both the execution time and energy consumption. Although, high accuracy is desirable, in many cases physical hardware measurements are difficult to deploy and use. Existing works demonstrated that energy modeling and estimation techniques could accurately estimate both execution time and energy consumption for embedded architectures similar to the ones used in this paper [14, 15]. Such estimation techniques can replace the physical-hardware measurements used in our approach in order to make the proposed technique accessible to more software developers.

5 Conclusion

Finding optimal optimization configurations for a specific compiler, architecture, and program is an open challenge since the introduction of compilers. Standard optimization levels that are built-in to modern compilers, on average perform well on a range of architectures and programs and provide convenience to the software developer. Over the past years, iterative compilation and complex machine learning approaches have been exploited to yield optimization configurations that outperform these standard optimization levels. These techniques are typically expensive either due to their large training phases or the large number of configurations that they need to test. Moreover, they are hardly portable to new architectures and compilers.

In contrast, in this work an inexpensive and easily portable approach that generates and tests less than 64 optimization configurations proved able to achieve execution-time and energy-consumption savings in the same range as the ones achieved by state of the art machine learning and iterative compilation techniques [7, 8, 22]. The effectiveness of this simple approach is attributed to the fact that we used subsequences of the optimization passes defined in the standard optimization levels, but stopped the optimizations at an earlier point than the standard optimization level under exploitation. This indicates that the accumulated empirical knowledge built into the standard optimization levels is a good starting point for creating optimization configurations that will perform better than the standard ones.

The approach is compiler and target independent. Thus, for its validation, two processors and two versions of the LLVM compiler framework were used; namely, the Arm Cortex-M0 with the LLVM v3.8 and the Arm Cortex-M3 with the LLVM v5.0. An average execution time reduction of 2.4% and 5.3% was achieved across all the benchmarks for the Cortex-M0 and Cortex-M3 processors, respectively, with at least half of the 71 benchmarks tested seeing performance and energy consumption improvements. Finally, our approach can support multi-criteria optimization as it targets execution time, energy consumption and code size at the same time.

In future work, clustering and other machine learning techniques can be applied on the compilation profiles retrieved by our exploitation approach (Figure 3) to fine-tune the standard optimization levels of a compiler to perform better for a specific architecture. Furthermore, the technique is currently being evaluated on more complex architectures, such as Intel's X-86.

Acknowledgments

The authors would like to thank Dr. Zbigniew Chamski for his valuable comments and helpful suggestions. The work is supported by the European Union's Horizon 2020 Research and Innovation Programme under Grant agreement No.: 779882, TeamPlay (Time, Energy and security Analysis for Multi/Many-core heterogeneous PLAtforms), and from the Royal Society Newton Advanced Fellowship Programme under Grant No.: NA160108.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, Washington, DC, USA, 295–305. https://doi.org/10.1109/CGO.2006.37

[2] ARM. 2018. Arm Cortex-M0 Processor. (2018). Retrieved February 19, 2018 from https://developer.arm.com/products/processors/cortex-m/cortex-m0

[3] ARM. 2018. Arm Cortex-M3 Processor. (2018). Retrieved February 19, 2018 from https://developer.arm.com/products/processors/cortex-m/cortex-m3

[4] ARM. 2018. Processors Cortex-M Series. (2018). Retrieved February 19, 2018 from https://www.arm.com/products/processors/cortex-m

[5] Amir H. Ashouri. 2016. *Compiler autotuning using machine learning techniques*. Ph.D. Dissertation. Politecnico Di Milano, Department of Computer Science and Engineering.

[6] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* 14, 3, Article 29 (Sept. 2017), 28 pages. https://doi.org/10.1145/3124452

[7] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *ArXiv e-prints* (jan 2018). arXiv:cs.PL/1801.04405

[8] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2015. A logic programming approach to predict effective compiler settings for embedded software. *Theory*

and Practice of Logic Programming 15, 4-5 (2015), 481–494. https://doi.org/10.1017/S1471068415000174

[9] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*. Paris, France. https://hal.inria.fr/inria-00475919

[10] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 185–197. https://doi.org/10.1109/CGO.2007.32

[11] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger, and M. Rosendahl. 2016. ENTRA: Whole-systems energy transparency. *Microprocess. Microsyst.* 47, Part B (Nov. 2016), 278–286. https://doi.org/10.1016/j.micpro.2016.07.003

[12] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (01 Jun 2011), 296–327. https://doi.org/10.1007/s10766-010-0161-2

[13] K. Georgiou, S. Xavier de Souza, and K. Eder. 2017. The IoT energy challenge: A software perspective. *IEEE Embedded Systems Letters* PP, 99 (2017), 1–1. https://doi.org/10.1109/LES.2017.2741419

[14] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. 2017. Energy Transparency for Deeply Embedded Programs. *ACM Trans. Archit. Code Optim.* 14, 1, Article 8 (March 2017), 26 pages. https://doi.org/10.1145/3046679

[15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static Analysis of Energy Consumption for LLVM IR Programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES '15)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2764967.2764974

[16] Simon Hollis. 2013. The MAGEEC energy measurement board. (Aug. 2013). Retrieved January 29, 2018 from http://mageec.org/wiki/Power_Measurement_Board

[17] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`.

[18] LLVMorg. 2018. LLVM's Analysis and Transform Passes. (2018). https://llvm.org/docs/Passes.html

[19] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 245–256. https://doi.org/10.1109/CGO.2017.7863744

[20] James Pallister. 2015. Pyenergy: An interface to the MAGEEC energy monitor boards. (Feb. 2015). Retrieved January 29, 2018 from https://pypi.python.org/pypi/pyenergy

[21] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2013. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *CoRR* abs/1308.5174 (2013). arXiv:1308.5174 http://arxiv.org/abs/1308.5174

[22] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2015. Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms. *Comput. J.* 58, 1 (2015), 95–109. https://doi.org/10.1093/comjnl/bxt129

[23] GCC team. 2018. GCC, the GNU Compiler Collection. (2018). Retrieved February 23, 2018 from https://gcc.gnu.org/