New Optimization Sequences for Code-Size Reduction for the LLVM Compilation Infrastructure

Anderson Faustino da Silva
UEM
Maringá, Paraná, Brazil
anderson@din.uem.br

Edson Borin
UNICAMP
Campinas, São Paulo, Brazil
edson@ic.unicamp.br

Fernando M. Quintão Pereira
UFMG
Belo Horizonte, Minas Gerais, Brazil
fernando@dcc.ufmg.br

Otávio Oliveira Nápoli
UNICAMP
Campinas, São Paulo, Brazil
onapoli@lmcad.ic.unicamp.br

Vanderson Martins do Rosário
UNICAMP
Campinas, São Paulo, Brazil
vrosario@lmcad.ic.unicamp.br

ABSTRACT

Typical compilers provide users with default optimization levels: sequences of code-transformation passes that can be tuned to either generate faster or smaller executables. Regarding this last dimension of efficiency—size—clang contains two optimization levels: `Os` and `Oz`. Both cause the invocation of a large number of passes: 264 and 260, respectively. In this paper, we explore the following question: is it possible to find shorter sequences that deliver similar results? To provide an answer to such a question, we have compiled 15,000 different programs with 10,044 optimization sequences, each with up to 100 code-transformation passes. Promising sequences, once selected, were then systematically reduced. From this effort, we have chosen five optimization sequences containing from 12 to 15 code transformations. When applied onto SPEC CPU2017, these sequences yield compilation times that are $1.4 - 1.6\times$ and $1.3 - 1.5\times$ faster than the `Os` and `Oz` optimization levels of LLVM. Moreover, one of the selected sequences produced code that is 2.3% smaller than LLVM's `Os` level, and only 2.5% larger than the `Oz` level.

CCS CONCEPTS

• **Software and its engineering** → **Runtime environments**; **Compilers**; *Software libraries and repositories.*

KEYWORDS

Benchmark, Optimization, Compiler, Search

ACM Reference Format:
Anderson Faustino da Silva, Edson Borin, Fernando M. Quintão Pereira, Otávio Oliveira Nápoli, and Vanderson Martins do Rosário. 2021. New Optimization Sequences for Code-Size Reduction for the LLVM Compilation Infrastructure. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3475061.3475085

Fernando Pereira has been supported by CNPq (Grant 406377/ 2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt). Edson Borin has been supported by Fapesp (2013/08293-7) and CNPq (Grant 314645/2020-9).

SBLP'21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9062-0/21/09…$15.00
https://doi.org/10.1145/3475061.3475085

1 INTRODUCTION

Compilers typically provide developers with a few *default optimization sequences*, each tuned for a different goal such as code size or speed. As an example, Opt, LLVM's middle-end optimizer, provides three default optimization sequences tuned for speed: `O1`, `O2`, and `O3`, plus two sequences tuned for code reduction: `Os` and `Oz`. These sequences involve a rather large number of *compilation passes*, *i.e.*, code analyses or code transformation routines. LLVM's `O1` level causes the invocation of 229 passes; `O2` invokes 277, and `O3`, 281. The code-size reduction sequences are no less frugal: Opt `-Os` accounts for 264 passes, and Opt `-Oz` invokes 260. Because these sequences are large, and not necessarily optimal, much effort has been made to improve them, be it via reordering their passes [10, 13], be it via stripping them away [5, 6, 14].

Purini and Jain [13] have observed that, depending on the number of default optimization levels available in a compiler, it is practical to apply all of them in a program, keeping the best result. From this observation, Purini and Jain introduced the notion of a *default cover set*: a collection of optimization sequences meant to be applied exhaustively onto the input program. A few years later, da Silva et al. [4] showed that it is relatively easy to engineer new optimization sequences that outperform those originally proposed by Purini and Jain. Nevertheless, da Silva et al. use a table with slightly more than 1,000 optimization sequences. Applying all of them onto the same program would still require a prohibitively long time for non-trivial programs.

The Goal of This Paper. This paper shares Purini and Jain's view, namely "*[A compiler should have] a set of few good sequences, so that for every class of programs there exists a good optimization sequence in the set catering to that class.*" This shared vision entices a common goal: the discovery of a small set of optimization sequences that can be applied exhaustively onto an input program, so as to keep the best result. However, that is as long as similarities go. We depart from Purini and Jain's work in a number of fundamental ways. First, whereas they focus on speed, we chose code size as the objective function. Our choice is pragmatic: speed fluctuates across samples; size does not. Thus, by focusing on code size, we can disregard a number of protocols that would be necessary for the sake of

statistical rigour. Furthermore, our study draws benchmarks from a collection of over one million programs [5]—a test bed that was not available when Purini and Jain performed their studies.

Nevertheless, the main difference between Purini and Jain's work and ours is methodological: instead of relying on clusterization to find sequences likely to be good on classes of programs, we decided to choose sequences with predetermined properties. We have selected five properties of interest—the subject of Section 3.2. Each one of them leads to a particular optimization sequence, which appear in Table 1. Section 4 evaluates these sequences.

Concrete Results. The new sequences are very effective: when combined, they produce code that is only 1.6% larger than LLVM's default `Oz` sequence. Also, if used individually, one of them (**Lim**) is capable of producing code that is 2.5% larger than the `Oz` sequence at 0.625× of the compilation time. Finally, in some cases, they outperform LLVM by a large margin. For instance, when compiling SPEC CPU2017's `x264`, our sequences lead to code 10% smaller. These results stem from the following contributions:

- **Exploration** This paper is one of the largest exploratory studies of the space of optimization sequences for code-size reduction. As we explain in Section 4.5, by using 15,000 benchmarks and 10,044 optimization sequences per benchmarks, we go beyond what had been previously accomplished, even by very recent work [4].
- **Criteria** Section 3.2 discusses the different criteria that we have adopted to choose good optimization sequences. We speculate that these properties are also meaningful in compilers different from LLVM, or even when applied onto programming languages other than C.
- **Sequences** The five sequences that Section 3.3 reports are very effective to reduce the size of binary programs. Furthermore, they are small enough to be used in settings where compilation time matters.

Software. The tools used to carry out the experiments discussed in this paper are available at https://github.com/ComputerSystemsLab/YaCoS, under the Apache-2.0 license.

2 OVERVIEW

Compilers apply code transformations onto programs to improve their performance or memory footprint. These transformations, known as optimizations, can be combined in several ways. The set of every possible combination of optimizations forms what we call the *Optimization Space*. This notion is well-known in the literature, but for the sake of completeness, Definition 2.1 restates it.

Definition 2.1 (Optimization Space). A sequence of optimizations S_v is said to be *valid* if, when applied on any program P, S_v produces a transformed program P' that is semantically equivalent to P, *i.e.*, given the same inputs, P and P' produce the same outputs. A valid sequence of optimizations S_m is said to be *meaningful* if, there exists a program Q, such that when S_m is applied onto Q, it yields a different program Q'. The set of every meaningful optimization sequence of a compiler forms this compiler's optimization space.

The optimization space is infinite, as already discussed in previous work [14]. To shield developers from the task of always looking for good optimization sequences, compilers typically provide them with a few default sequences, as Example 2.2 illustrates.

Example 2.2. The LLVM compilation infrastructure defines three optimization levels tuned for speed: `O1`, `O2` and `O3`. It also defines two optimization levels tuned for size: `Os` and `Oz`. Additionally, LLVM provides the `O0` level, which disables as many optimizations as possible. Yet, some optimizations still run. As an example, the inliner can be invoked upon functions annotated with the `always_inline` modifier. The `Os` level runs almost the same ensemble of optimizations than the `O2` level. However, it uses a cost model that focus on code size, instead of code speed. The `Oz` sequence is more aggressive than `Os`, limiting, for instance, the amount of unrolling and the size of functions that can be inlined[1].

Such sequences are built out of the knowledge of expert engineers, and tend to enjoy the scrutiny of a large community of users. Nevertheless, it is unlikely that one optimization sequence will be good for every program. Appel's *Full-Employment Theorem* [1] makes it impossible to devise a single sequence that outperforms all the others[2]. In addition of being suboptimal, the default sequences of LLVM tend to present many redundancies. Example 2.3 discusses redundancies in the two sequences used by LLVM to reduce code size: `Os` and `Oz`. Notice, nevertheless, that although these two sequences are very similar, they still use different cost models—a series of constants that determine how the optimizations behave. The cost model guides the code transformations applied by the optimizations. Such differences are more evident in optimizations that bear higher impact on code size, such as loop unrolling and inlining.

Example 2.3. The default size-reduction levels, `Os` and `Oz`, cause the invocation of 264 and 260 passes, respectively. Within these sequences, we find a common subsequence with 259 elements! In other words, except for one instance of loop transformation, `Oz` is a subset of `Os`[3]. The latter, in turn, performs SLP Vectorization (Superword-Level Parallelism), an optimization that is absent on the former.

The different optimization levels used in LLVM have been built along many years of experimentation, by seasoned compiler engineers. Our perception of this building process is that the LLVM community prefers to increase them by adding new optimizations, as these new passes become available. Optimizations are seldom removed from these sequences, unless the pass that implements them becomes obsolete. This perception is unfortunate, as previous work has already shown that it is possible to obtain better code by eliminating some optimizations from default sequences [9]. It is our view that LLVM, or any ahead-of-time optimizing compiler, would gain much from providing developers with a more diverse suite of optimization sequences. In the next section, we introduce

[1] In the specific case of LLVM v10.0.0, more information about the different optimization levels that this compiler uses, including the constants that define the cost model, can be found in the `PassBuilder.h` header file.

[2] The smallest program that does nothing and does not terminate is P_{least} = `L: goto L;` The perfect compiler, when fed with a program that does not generate output, and does not terminate, must produce P_{least}. Such compiler would, thus, solve the Halting Problem.

[3] LLVM's `Oz` still perform all the loop transformations used in `Os`, albeit in a slightly different order.

a methodology to build such sequences, aiming at removing unwanted redundancies, while still producing good-quality binary code.

3 SYSTEMATIC SEARCH

This section explains how we have explored LLVM's optimization space in search of sequences for code-size reduction. This exploration demands a large number of benchmarks. As already observed in previous work, more than 1,000 different programs are required, on average, for a random sequence of 100 optimizations to outperform `opt -Os` with more than 10% of chance [14]. To obtain a large number of different programs, we use AnghaBench, a collection of over one million C functions mined from open-source repositories [5]. Functions are provided as independent compilation unit: C files with a single function plus all the declarations to render it compilable[4]. We chose to use the 15,000 largest functions available in AnghaBench. This choice is pragmatic: ideally, we should use as many functions as possible; however, the methodology that we shall present in Section 3.1 requires substantial time and computational power. In total, it took us six weeks to build the optimization matrix, a notion yet to appear in this paper (Definition 3.1).

3.1 Search Methodology

The optimization space (Definition 2.1) is infinite, as already explained in Section 2. Thus, searching it for good optimization sequences requires some form of downsampling. In this paper, we adopt the following methodology to downsample the optimization space:

(1) Generate optimization sequences for benchmark, using the genetic algorithm provided by YaCoS [17]. In this algorithm, individuals are numeric vectors, where each position contains the index of an analysis or optimization pass. For each benchmark, we have set a population of 100 individuals, with 100 genes, and have evolved it over 50 generations. The crossover probability is 0.9 and the mutation probability is 0.1.
(2) Reduce the best sequence (for every program), via Purini and Jain's reduction algorithm [13]. Such algorithm removes optimizations from a sequence, as long as this optimization does not compromise the performance of the sequence.
(3) Filter out duplicates from the reduced sequence set. Two sequences are considered duplicates if they have the same optimizations, in the same order.

After removing duplicates, we were left with 10,044 sequences of various sizes. From this set of 15,000 programs, and 10,044 sequences, we build an *optimization matrix*, a notion that we define as follows:

Definition 3.1 (Optimization Matrix). The optimization matrix M is a table of programs $\times$ sequences. An entry (P_i, S_j) in M represents the number of instructions that sequence S_j produced when applied onto program P_i.

[4]One limitation of using single functions is that the impact of inlining and other interprocedural optimizations cannot be evaluated.

3.2 Properties of Interest

The main goal of our optimization sequences is to minimize the size of the generated code. Nonetheless, there are several ways of comparing these sequences when analyzing their effect on a set of 15,000 programs. Moreover, depending on the property of interest, one optimization sequence may be better than the other. The following list introduce the main properties of interest we used to compare the optimization sequences. Any of these properties can be directly mined from the optimization matrix built in Section 3.1.

- **Best frequency (Bst)**: this property indicates how often the optimization sequence produced the best code, *i.e.*, a code that is the smallest among the code produced by the 10,044 sequences. To derive this metric, we count how often a sequence S_i was the minimum per line of the optimization matrix, and returns the most frequent winner.
- **Good in bounds (Lim)**: this property indicates how often the optimization sequence produced a code that is considered good, *i.e.*, it is at most X times (*e.g.*, 1.03×) larger than the best code. We explored three different thresholds to define what would be considered good (1.03, 1.05, and 1.07). This metric is particularly useful when the difference in the code size generated by a sequence, in relation to the code size generated by best sequence, is small, since it discards this small variation.
- **Best total size (Sum)**: we define the optimization sequence total code size as the sum of the sizes of all the 15,000 codes produced by the optimization sequence. To compute this metric, we sum up the columns of the optimization matrix, and return the smallest sum.
- **Relative-size geometric mean (Geo)**: the relative code size indicates how much larger is the code produced by the optimization sequence when compared to the smallest code produced among all sequences. For example, a relative code size of 1.35× indicates that the code produced by the optimization sequence for a given program is 1.35 times larger than the best code produced among all sequences. We employ the geometric mean to summarize the relative code size of all programs for a given optimization sequence.
- **Best maximum relative size (Cap)**: The maximum relative size is the worse relative size achieved by an optimization sequence for all program. The best maximum relative size is the smallest maximum relative size. In other words, the optimization sequence that produces the best maximum relative size, is the one that produces the least-worse maximum relative size.

In Section 3.3, we describe the best sequences obtained according to each property.

3.3 Five Sequences

Table 1 shows the best sequences found according to each property of interest. The second column reports the optimizations, in order, that constitute the sequence and the third reports the total number of passes invoked by those optimizations. Notice that the number of optimizations and the number of passes invoked differ. Differences are due to dependencies between passes. For instance, any sequence of optimizations must start with `-targetlibinfo` and

`-tti`. Similarly, any sequence must end with `-verify`. Therefore, if the defined sequence does not have these passes, they are added automatically.

Seq.	Passes	Total
Bst	mem2reg, jmp-threading, instcombine, early-cse-memssa, jump-threading, licm, early-cse-memssa, sroa, simplifycfg, reassociate, instcombine, slp-vectorizer, early-cse-memssa	57
Lim	sroa, early-cse-memssa, reassociate, instcombine, simplifycfg, licm, speculative-execution, jump-threading, early-cse-memssa, simplifycfg, instcombine, simplifycfg	56
Sum	mem2reg, early-cse-memssa, correlated-propagation, instcombine, reassociate, simplifycfg, early-cse-memssa, instcombine, licm, jump-threading, simplifycfg, dse, reassociate, early-cse-memssa, instcombine	40
Geo	loop-vectorize, sroa, gvn, instcombine, simplifycfg, instcombine, licm, gvn, correlated-propagation, jump-threading, mldst-motion, early-cse-memssa, instcombine, simplifycfg, instsimplify	79
Cap	loop-rotate, sroa, correlated-propagation, indvars, gvn, tailcallelim, instcombine, jump-threading, reassociate, simplifycfg, instcombine, early-cse-memssa	58

Table 1: The best sequences. The second column reports the passes found by our methodology, and the third column reports the total number of passes invoked by the sequence.

On the Performance of the Chosen Optimization Sequences. Table 2 summarizes the results achieved by the selected sequences according to each property of interest. The first row shows that the **Bst** sequence produced the smallest code in 4,161 benchmarks, a number that corresponds to 27.74% of all programs. The **Lim** sequence produced code sizes that are up to 1.03/1.05/1.07 times larger than the smallest code for 11,012, 12,712 and 13,546 programs. These numbers correspond to 73.4%, 84.7% and 93.8% of all benchmarks. The **Sum** sequence produced, in total, 3 295 639 instructions for all 15,000 programs. Instructions, in this case, correspond to LLVM bytecodes. The **Geo** sequence achieved the best relative-size geometric mean. When compared to the best sequence for each available program, on average, **Geo** is only 1.0324 times worse. Finally, the **Cap** sequence produced the least-worse relative-size. When compared to the best sequence for a particular benchmark, it produced code that was 16.31 times larger. For all the other sequences it was possible to find worse relative results.

Table 2 also shows, for each sequence, its rank according to each property of interest. For example, the **Bst** sequence achieved the best result (rank #1) according to the "Best Frequency" property of Section 3.2. It got the second best result (rank #2) on the "Good in bounds" property, in this case with a threshold of 1.03. The **Lim** sequence achieved very good results (ranked 1^{st} or 2^{nd}) in most properties of interest. In fact, the only property of interest for which this sequence did not perform well is the Best Maximum Relative Size (last column). The **Bst**, **Sum**, and **Geo** sequences also achieved good or moderate results on most of the properties of interest, the exception being, again, the Best Maximum Relative Size. The **Cap** sequence, on the other hand, offers the best Maximum Relative Size but achieved poor results on the other properties of interest. We emphasize that these sequences can—in specific benchmarks—greatly outperform the default optimization levels used in LLVM. Experiments in Section 4 corroborate this claim.

4 EVALUATION

In this section, we evaluate the efficacy of the different sequences described in this paper. To this end, we shall provide answers to the following questions:

- **RQ1:** What is the effectiveness of the new proposed sequences, in terms of code-size reduction?
- **RQ2:** What is the effectiveness of the new proposed sequences, in terms of compilation time?
- **RQ3:** How diverse are the new sequences?
- **RQ4:** What is the added benefit of combining all the sequences?
- **RQ5:** How does the sequences compare to the *Default Cover Set* introduced by Silva et al. [14]?

Software: Optimizations are carried out via LLVM version 10.0.0. The operating system used was Debian 10. Sequences are evaluated on the 16 programs from SPEC CPU2017. Hence, although our sequences have been trained on individual functions from AnghaBench, they are tested onto the whole programs from SPEC CPU2017. The scripts to run these benchmarks have been adapted from AutoParBench [12].
Hardware: Every experiment discussed in this paper has been performed on an Intel Xeon E5-2650 (2.30 GHz, 32G RAM, with 12 cores and hyper-threading).

4.1 RQ1 – Code-Size Reduction

The different sequences evaluated in this paper have been selected based on their ability to reduce code size. To measure this ability, we compare them with the default code-size reduction sequences used by LLVM. We emphasize that these default sequences are formed by a very large number of code analyses and transformation passes: LLVM's `Opt -Os` runs 264 passes while `Opt -Oz` runs 260 passes. Our sequences, in contrast, invoke about 58 passes on average. Therefore, we do not expect to consistently beat the default optimization sequences of LLVM, although victories could be achieved. This section reports these results.
Methodology. RQ1 is evaluated on the programs in SPEC CPU2017. Each program is composed of several compilation units (C files). We report code size as the number of LLVM instructions in the program after linking takes place. Thus, every file that constitutes a benchmark is compiled with the same optimization sequence. The resulting bytecodes are linked into a single file, whose size, in number of LLVM instructions, is logged.
Discussion. Figure 1 shows the numbers that sprout from the above methodology. We report the sizes obtained with the five

Seq.	Results for each property of interest - Rank (value)						
	Best Frequency	Good in bounds			Total Size	Rel. Size Geomean	Best Max. Rel. Size
		≤ 1.03 × best	≤ 1.05 × best	≤ 1.07 × best			
Bst	**#1 (4161)**	#2 (10993)	#2 (12655)	#7 (13453)	#45 (3306270)	#70 (1.0372)	#7610 (455.00)
Lim	#2 (4097)	**#1 (11012)**	**#1 (12712)**	**#1 (13546)**	#2 (3295888)	#2 (1.0325)	#539 (69.00)
Sum	#28 (3873)	#6 (10783)	#3 (12583)	#3 (13511)	**#1 (3295639)**	#9 (1.0344)	#1558 (132.00)
Geo	#282 (3446)	#14 (10671)	#5 (12574)	#2 (13544)	#4 (3298863)	**#1 (1.0324)**	#522 (68.50)
Cap	#2774 (2431)	#3787 (7412)	#3775 (9948)	#3687 (11627)	#2896 (3355929)	#2089 (1.0502)	**#1 (16.31)**

Table 2: Overall performance of selected sequences. Each cell shows the rank number and the performance of a given sequence (row) with respect to a given property of interest (column).

Bench	c-O0	o-Os	o-Oz	bst	lim	sum	geo	cap	comb
perlb	713428	354359	343717	344439	343996	346648	347050	353159	341877
gcc	2725310	1358541	1314075	1341738	1335201	1351872	1346199	1370327	1326314
mcf	6977	3944	3005	3105	3117	3132	3079	3172	3051
nam	255612	142230	112520	112352	111595	111842	111933	119507	111174
parest	1957921	1073925	1014315	1047082	1025827	1027513	1024170	1051729	1015260
povray	236209	127226	121697	123056	122846	123644	123189	125167	121064
lbm	7783	2597	2634	3006	2969	2996	2973	2610	2607
omnet	372492	200542	197040	203690	202808	203091	203001	204889	201866
xalanc	886714	473507	460120	473235	472228	472784	472123	477474	469507
x264	163732	84846	80129	72216	72183	72191	72365	73028	71136
blender	2882043	1375976	1312800	1388703	1384293	1398034	1386408	1420129	1372684
deepsj	26625	14654	13616	13497	13525	13642	13618	13828	13387
imagick	482423	244221	228469	244257	242167	242486	241721	249225	239787
leela	47831	24737	23552	24302	23553	23658	23526	23662	23343
nab	51828	27780	25240	26727	26728	26693	26866	27795	26367
xz	51139	25406	24252	24428	23737	23734	23787	24178	23423
Total	10868067	5534491	5277181	5445833	5406773	5443960	5422008	5539879	5362847

Figure 1: Number of instructions in the final, linked, program, produced by different optimization sequences. Winning strategies are highlighted.

proposed optimization sequences, plus sizes achieved by `clang -O0` (`c-O0`), `opt -Os` (`o-Os`), and `opt -Oz` (`o-Oz`). In addition to results obtained with the five sequences from Section 3, Figure 1 also shows the size of programs compiled with all the sequences combined. We refer the reader to Section 4.4 for an explanation on how combinations were produced. The combination of sequences is able to outperform the default sequences of LLVM in seven, out of 16 programs. The default sequences still outperform the majority of the sequences, when they are tried individually. A noticeable exception is observed in `x264`, in which **Sum** and **Lim** outperform `opt -Oz` by 11%, and `opt -Os` by 16%. Nevertheless, most of the results are close, and all the sequences considered in this study are able to generate code that is, in general, less than half the size of the unoptimized code (code produced with `clang -O0`). Finally, the **Lim** sequence produces code that is 2.3% smaller than Opt `-Os` and only 2.5% larger than Opt `-Oz`.

4.2 RQ2 – Compilation Time

We define as *compilation time* the time taken to invoke the LLVM optimizer (Opt) with a given sequence of optimizations on the input program. Therefore, this time includes not only the time to apply the optimizations, but also time spent in parsing the LLVM IR, and dumping the optimized program.

Methodology. To measure the compilation time of a given optimization sequence, we run Opt five times with that sequence over all the programs in SPEC CPU2017. If t_s is the sum of the time taken to optimize all the programs, then we report the average of five measurements of t_s.

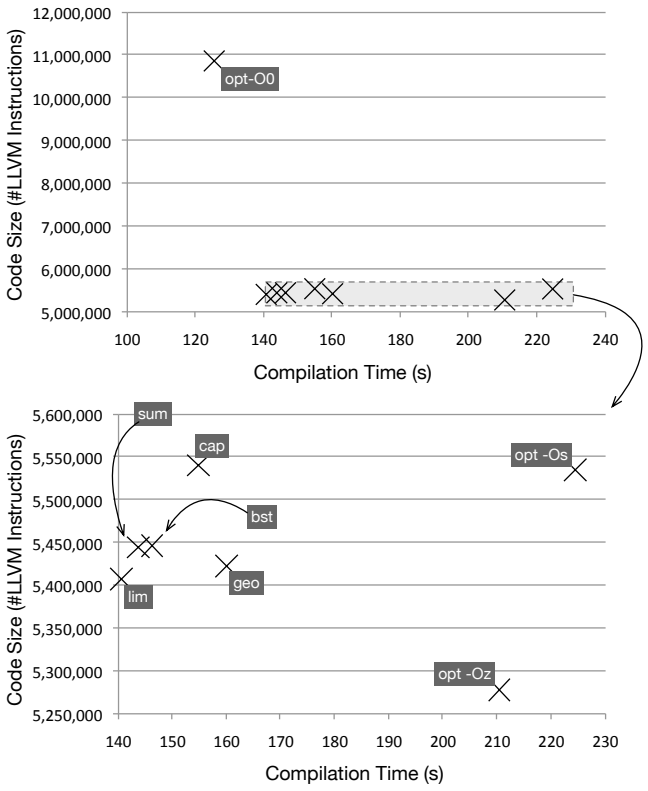

Figure 2: Comparison between the effectiveness and the running time of different code optimization sequences when optimizing applications from the SPEC CPU2017 benchmark.

Discussion. Figure 2 shows the compilation time of the different sequences. The default sequences used in LLVM, `Os` and `Oz` take considerably more time than the others: 224.0 and 210.0 seconds. The fastest sequence is **Lim**, even though it is not the shortest sequence: it invokes 56 LLVM passes, whereas **Sum** invokes 40. The application of **Lim** on SPEC CPU2017 takes 140.0 seconds. Nevertheless, the difference is small: the slowest among our sequences is **Geo**, which takes 160.0 seconds. Consequently, our sequences

are 1.4 − 1.6× faster than `0s` and 1.3 − 1.5× faster than `0z`. For reference purposes, Figure 2 also shows the size of the programs, in number of LLVM instructions, produced by the different sequences. These results are discussed in Section 4.1. Interestingly, the fastest sequences also yield the smallest programs. This observation is valid for the relation between all the sequences, except for **Geo**, which, being the slowest, produces the second smallest binaries.

4.3 RQ3 – Diversity

As already seen in Example 2.3, Page 2, the default optimization levels that LLVM uses to reduce code size, `0s` and `0z`, are very similar. In contrast, the sequences explored in this paper show greater diversity. In this section, we provide data that supports this statement. To this end, we evaluate the effect of the many sequences hitherto considered when applied onto a large universe of programs.

Methodology. To provide an idea of "distance" between two different sequences, we shall evaluate the effects of these sequences onto a large set of programs. In this section, we consider the same collection of 15,000 programs previously used in systematic search (Section 3). Thus, for each optimization sequence s, we derive a vector v_s of 15,000 dimensions. The i^{th} position of v_s represents the *relative* performance of s on f_i, the i^{th} program in the program collection. The relative performance is measured as the size of the code produced by s when compiling f_i divided by the size of f_i when compiled by the *best* sequence for f_i, *i.e.*, the sequence that produces the smallest code for f_i. To obtain the best sequence for f_i, we apply all the 10,004 sequences considered in Section 3 onto f_i, keeping the best result. Therefore, every position of v_s is a number in the interval $[1.0, +\infty]$. To obtain the distance between two sequences, s and t, we compute the Euclidean distance between v_s and v_t on a 15,000-dimensional space.

	sum	cap	bst	geo	lim	c-Os	c-Oz	o-Os	o-Oz
sum	0	183.2	420.7	158.3	171.7	173.8	172.7	176.5	173.1
cap	183.2	0	493.2	99.4	73.9	258.7	261.1	257.5	256.7
bst	420.7	493.2	0	483.3	487.7	382.3	381.8	380.3	378.7
geo	158.3	99.4	483.3	0	66.4	262.7	261.9	260	257.6
lim	171.7	73.9	487.7	66.4	0	254.2	253.5	251.3	248.9
c-Os	173.8	258.7	382.3	262.7	254.2	0	12.6	31.6	26.5
c-Oz	172.7	261.1	381.8	261.9	253.5	12.6	0	32.6	23.8
o-Os	176.5	257.5	380.3	260	251.3	31.6	32.6	0	21.9
o-Oz	173.1	256.7	378.7	257.6	248.9	26.5	23.8	21.9	0

Figure 3: Heat map comparing the Euclidean distance between the different sequences considered in this paper. We let `c-0s` denote `clang -0s`, and `o-0s` denote `opt -0s`. Similar representation is used for `0z`.

Discussion. The symmetric matrix in Figure 3 shows the result of this experiment. The darker the color of the cell, the more different are the results produced by the two sequences compared by that cell. From the figure, a number of facts are evident. First, the default optimization levels of LLVM produce similar results. Even though they rely on different cost models, they still use almost identical sequences of passes. The ordering of passes within these sequences still seems to bear the strongest impact on the effects of an optimization level. This result is still valid if we consider either `clang` or `opt` as the driver of optimizations[5]. The five sequences described in this paper, in contrast, show much greater diversity, be it when we compare them among themselves (upper-left corner of Figure 3), be it when we compare them with LLVM's default optimization levels (bottom-left or upper-right corners of Figure 3). In particular, **Bst** produces effects that are quite dissimilar to those observed via the other sequences considered in this paper.

4.4 RQ4 – Combined Usage

The five sequences discussed in Section 3 have been obtained via testing on a large collection of single-function programs. As discussed in Section 4.3, these sequences perform very differently in these programs. Yet, the evaluation carried out in this section applies them onto programs that contain multiple functions. It is thus possible that some of the gains that a sequence obtains in some functions be cancelled out in others. Therefore, it makes sense to use the sequences as a cover set, meaning that they are all applied onto each function that constitutes the target program, the best results being kept throughout this process. In this section, we evaluate the gains of such an approach.

Methodology. LLVM does not support the application of different sequences of optimizations onto different functions that belong to the same program. In other words, all the functions within a program module, *i.e.*, a program's object file, must be compiled with the same sequence of passes. Therefore, to answer **RQ4**, some engineering effort was required. To apply all the optimization sequences onto each function of a program, we split said program into independent files—one file per function. Each file is compiled five times: once per sequence. The best result, measured as the smallest object file in number of LLVM instructions, is then kept. These object files are, finally, linked into a final binary, which is the product of this compilation pipeline. This rather convoluted process turns any attempt to measure compilation time meaningless: most of the processing time is spent into input and output operations: creating, writing and merging binary files. As a result, the time to compile SPEC CPU2017 takes 15× longer than the process of compiling each program five times, once with each one of the five sequences.

Discussion. The last column in Figure 1 shows the size of the final binary produced by the combination of all the sequences. This combination outperforms the default optimization levels of LLVM in seven, out of 16 benchmarks. An immediate question is: why not more? Firstly, the default sequences are much longer than the five new sequences. Therefore, they are likely to naturally include the effects of many of them in several different benchmarks. Second,

[5] `Clang -Oz` and `opt -Oz` do not yield the same effect onto the final binary produced. The former adds flags such as `optsize`, `minsize` and `noinline` into the program's intermediate representation. These flags alter the cost model used by different optimizations available in the LLVM infrastructure.

our sequences are tried into individual functions; hence, we are unable to carry out interprocedural optimizations.

4.5 RQ5 – Default Cover Set

In 2013, Purini and Jain [13] introduced the notion of a *default cover set*: a small collection of optimizations that tend to be good for a given set of benchmarks. In addition to defining this concept, Purini and Jain showed how to build a cover set using heuristics for set cover. The original work of Purini and Jain would build a cover set using execution speed as the objective function. Years later, Silva et al. [14] showed how to adapt Purini and Jain's work to use code size instead of speed. Unfortunately, we cannot use the cover set discovered by Silva et al. directly, as they have used a different version of the LLVM compiler than the one we are using in this evaluation. Nevertheless, we could reuse their techniques to build a new cover set, and compare it with our five sequences.

Methodology. We build a default cover set using the collection of 15,000 programs used in Section 3. To build the set, we use the same heuristic adopted by Silva et al.. We refer the reader to their work for more details. The application of said heuristic in our collection yielded eleven optimization sequences, which we shall name `S0`, `S1`, . . ., `S9`, `SA`. We compare the cover set and the other sequences discussed in this paper using the package of benchmarks called `Angha10K`, which is distributed with the AnghaBench suite [5]. This package contains 10,000 programs of various sizes. These programs are, in principle, not present in the collection of 15,000 used in the search described in Section 3. We say "in principle" because AnghaBench is mined from open source repositories. Hence, it is possible that the same function is reused in two different repositories; however, we have not used any form of plagiarism detection to filter out repeated entries in our suite of benchmarks.

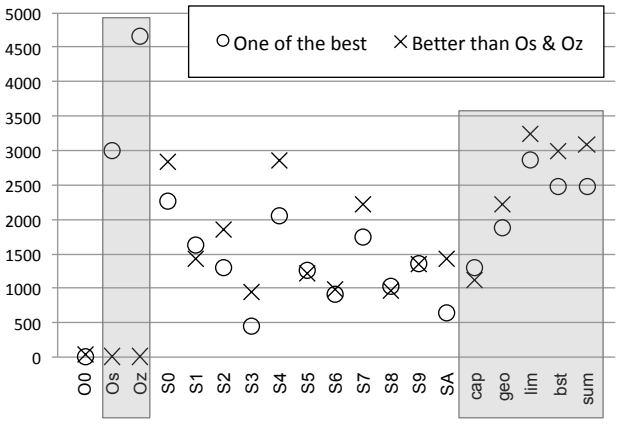

Figure 4: Comparison between the five sequences and ten sequences that yield a default-cover set, following the work of Silva et al. [14].

Discussion. Figure 4 shows the relative performance of the different optimization sequences in the cover set and the sequences discussed in Section 3. The figure also shows the performance of the default optimization levels of LLVM. We have ranked each sequence along two dimensions. The first, "One of the best", shows how often each sequence yielded the best results, counting ties. The second, "Better than Os & Oz" counts how often each sequence has been able to strictly outperform either of the default optimization levels of LLVM. Figure 4 shows that our three most effective sequences, **Lim**, **Bst** and **Sum** outperform any of the eleven sequences in the default cover set. They lose only to `opt -Oz`, a sequence more than five times larger then their average size. This experiment also reveals that the "Best Frequency" metric, used in Section 3 to build the sequence called **bst** is not stable: two of our sequences, **lim** and **sum**, have been able to outperform **bst** in this experiment.

5 RELATED WORK

The set of all the possible ways to combine the optimizations available in a compiler is called the compiler's *optimization space*. This paper explores the optimization space of LLVM, using code-size as a guiding objective function. The exploration of the optimization space of different compilers has been the goal of many previous research projects [5, 7, 8, 11, 13–15]. Much of the enthusiasm elicited by this goal comes from recent progress achieved in the field of research formed by the intersection between compilers and machine learning [2, 16]. The existence of public data mining tools, coupled with the current availability of benchmarks for compilers has greatly contributed to increase the effectiveness of methods to explore the optimization space of compilers.

The programming language community has been showing increasing interest in code-size reduction techniques [3, 5, 14]. However, much of the effort to explore the optimization space has been guided by speed, not code size, as the main driving force [7, 11, 13, 15]. An exception of notice to this trend is the recent work of Faustino *et al.* [5, 14], who have proposed ways to tune LLVM for code-size reduction. da Silva et al. try to predict good optimization sequences for a program, by comparing this program with a database of well-known codes. The basic idea of this line of work is to use similarity search to find, given an unknown program, functions that resemble it.

In the past, different researchers have proposed new optimization sequences for mainstream compilers, as alternatives to its default optimization levels. In the LLVM community, one of the first and most systematic approaches to craft new optimization sequences is due to Purini and Jain [13]. Purini and Jain have invented the notion of a *cover set*, that is, a small set of optimization sequences that is likely to be good for any program. The original work of Purini and Jain used execution speed as the objective function to build the cover set. Later, that very research group [10] took advantage of the same search infrastructure to analyze subsequences from clang -Os, aiming at finding smaller optimization sequences that would be still effective for code-size reduction. More recently, Silva et al. [14] reused Purini and Jain's ideas to find a cover set tuned to reduce code. The experiment discussed in Section 4.5 reuses Silva et al.'s publicly available artifact (https://zenodo.org/record/4416117).

6 CONCLUSION

This paper has described a methodology to find good optimization sequences for code size reduction. This methodology consists of a search procedure, plus five criteria. Each criteria, in our experiments, led to a different optimization sequence. Although these five

sequences cannot consistently outperform the default optimization levels of LLVM, they require less compilation time because they are much shorter. Additionally, they can produce code substantially smaller than `opt -Oz`, for some benchmarks. As an example, these sequences produce code that is 10% shorter than the code produced by `opt -Oz`, in about half the compilation time.

Our choice of criteria is arbitrary. Probably, different properties could be used to obtain optimization sequences that are still more effective than those that we study in this paper. The quest for such criteria, and the development of other methodologies to explore the optimization space of mainstream compilers is an ongoing project of ours. This paper is a step towards chartering this space. However, it is our vision that such exploration must be a community effort, given the sheer size of the space yet to be chartered.

REFERENCES

[1] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press, USA.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *Comput. Surv.* 51, 5 (2018), 96:1–96:42. https://doi.org/10.1145/3197978

[3] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production iOS Mobile Applications. In *CGO*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, New York, NY, USA, 363–377. https://doi.org/10.1109/CGO51591.2021.9370306

[4] Anderson Faustino da Silva, Bruno Kind, José Wesley Magalh aes, Jerônimo Rocha, Breno Guimar aes, and Fernando Magno Quint ao Pereira. 2020. *AnghaBench: a Synthetic Collection of Benchmarks Mined from Open-Source Repositories.* Technical Report 01-2020. Universidade Federal de Minas Gerais.

[5] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *CGO*. IEEE, Los Alamitos, CA, USA, 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[6] Tiago Cariolano de Souza Xavier and Anderson Faustino da Silva. 2018. Exploration of Compiler Optimization Sequences Using a Hybrid Approach. *Computing and Informatics* 37, 1 (2018), 165–185.

[7] Shuangde Fang, Wenwen Xu, Yang Chen, Lieven Eeckhout, Olivier Temam, Yunji Chen, Chengyong Wu, and Xiaobing Feng. 2015. Practical Iterative Optimization for the Data Center. *ACM Trans. Archit. Code Optim.* 12, 2, Article 15 (May 2015), 26 pages. https://doi.org/10.1145/2739048

[8] João Fabrício Filho, Luis Gustavo Araujo Rodriguez, and Anderson Faustino da Silva. 2018. Yet Another Intelligent Code-Generating System: A Flexible and Low-Cost Solution. *J. Comput. Sci. Technol.* 33, 5 (2018), 940–965. https://doi.org/10.1007/s11390-018-1867-7

[9] 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*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/3207719.3207727

[10] Shalini Jain, Utpal Bora, Prateek Kumar, Vaibhav B. Sinha, Suresh Purini, and Ramakrishna Upadrasta. 2019. An analysis of executable size reduction by LLVM passes. *CSIT* 7, 1 (2019), 105–110. https://doi.org/10.1007/s40012-019-00248-5

[11] Luiz G. A. Martins, Ricardo Nobre, João M. P. Cardoso, Alexandre C. B. Delbem, and Eduardo Marques. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Trans. Archit. Code Optim.* 13, 1, Article 8 (March 2016), 28 pages. https://doi.org/10.1145/2883614

[12] Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. 2020. AutoParBench: A Unified Test Framework for OpenMP-Based Parallelizers. In *ICS*. Association for Computing Machinery, New York, NY, USA, Article 28, 10 pages. https://doi.org/10.1145/3392717.3392744

[13] Suresh Purini and Lakshya Jain. 2013. Finding Good Optimization Sequences Covering Program Space. *ACM Trans. Archit. Code Optim.* 9, 4, Article 56 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400715

[14] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools. In *Compiler Construction*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/3446804.3446849

[15] John Thomson, Michael O'Boyle, Grigori Fursin, and Björn Franke. 2010. Reducing Training Time in a One-Shot Machine Learning-Based Compiler. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 399–407.

[16] Zheng Wang and Michael F. P. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[17] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. 2020. YACOS: A Complete Infrastructure to the Design and Exploration of Code Optimization Sequences. In *SBLP*. Association for Computing Machinery, New York, NY, USA, 56–63. https://doi.org/10.1145/3427081.3427089