AnghaBench: a Suite with One Million Compilable C Benchmarks for Code-Size Reduction

Anderson Faustino da Silva
Department of Informatics
UEM, Brazil
anderson@din.uem.br

Bruno Conde Kind
Department of Computer Science
UFMG, Brazil
condekind@dcc.ufmg.br

José Wesley de Souza Magalhães
Department of Computer Science
UFMG, Brazil
josewesleysouza@dcc.ufmg.br

Jerônimo Nunes Rocha
Department of Computer Science
UFMG, Brazil
jeronimonunes@dcc.ufmg.br

Breno Campos Ferreira Guimarães
Department of Computer Science
UFMG, Brazil
brenosfg@dcc.ufmg.br

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

***Abstract*—A predictive compiler uses properties of a program to decide how to optimize it. The compiler is trained on a collection of programs to derive a model which determines its actions in face of unknown codes. One of the challenges of predictive compilation is how to find good training sets. Regardless of the programming language, the availability of human-made benchmarks is limited. Moreover, current synthesizers produce code that is very different from actual programs, and mining compilable code from open repositories is difficult, due to program dependencies. In this paper, we use a combination of web crawling and type inference to overcome these problems for the C programming language. We use a type reconstructor based on Hindley-Milner's algorithm to produce AnghaBench, a virtually unlimited collection of real-world compilable C programs. Although AnghaBench programs are not executable, they can be transformed into object files by any C compliant compiler. Therefore, they can be used to train compilers for code size reduction. We have used thousands of AnghaBench programs to train YaCoS, a predictive compiler based on LLVM. The version of YaCoS autotuned with AnghaBench generates binaries for the LLVM test suite over 10% smaller than clang -Oz. It compresses code impervious even to the state-of-the-art Function Sequence Alignment technique published in 2019, as it does not require large binaries to work well.**

***Index Terms*—Benchmark, Repository, Synthesis, Training**

I. Introduction

The growing popularity of stochastic classification techniques is contributing to making compiler autotuning an effective approach to the generation of efficient programs [1], [2], [3]. Autotuning is implemented as follows. A compiler is trained on a collection of programs, and uses knowledge acquired during training to optimize unseen codes. Samples from the known collection of programs are compiled in different ways, and the best results, given some objective function such as running time or size, are recorded. The compiler uses this model to decide which optimizations to apply on an unknown program P_u. Autotuning has been shown to be effective along different dimensions of code efficiency, such as running time [4], [5], [6], [7], [8], energy consumption [9], [10], code size [11], [12], [13], hardware usage [14], [15], and the size-speed relation [16], [17], for instance.

a) The Problem Posed by the Lack of Benchmarks: A common shortcoming in this field, extensively discussed by Cummins et al. [14], is the small size of typical training sets. Cummins et al. have analyzed 25 research papers published between 2013 and 2016, from four conferences: CGO, HiPC, PACT, and PPoPP. They observe that "*the average number of benchmarks used in each paper [is] 17*". This result, although at first surprising, should not be unexpected. Typical benchmarks contain a small number of programs: SPEC CINT2006 [18] contains 12, SPEC CFP2006 [18] contains 17, Parsec [19] contains 13, Rodinia [20] contains 23, Polybench [21] contains 30, cBench [22] contains 30, and NPB v.1 [23] contains 8. The problem with these small numbers is, in the words of Cummins et al., that "*heuristics learned on one benchmark suite fail to generalize across other suites*". Wang and O'Boyle subsume well the essence of the problem: "*The most immediate problem continues to be gathering enough sufficient high quality training data. Although there are numerous benchmark sites publicly available, the number of programs available is relatively sparse compared to the number that a typical compiler will encounter in its lifetime.*" [3]

To circumvent the obstacle posed by a perceived lack of benchmarks, compiler researchers resort to program generation. With such purpose, automatically constructed programs have been used to tune compiler heuristics in specific scenarios [24], [10], [25], [26], [27]. However, these programs cannot be easily employed in general purpose compilers: they consist of micro-kernels that exercise particular aspects of the target hardware or of the target programming language. As an example, Sreelatha et al. [10] generate code snippets to find optimum constants for their code generation approach. Each program performs one action several times, be it to access memory, to synchronize threads, to force branch mispredictions, etc. Such behavior, although befitting Sreelatha et al.'s needs, is unlikely to occur in real-world programs.

b) Our Contributions: We bring forward a new technique to generate compilable benchmarks for C. As we explain in Section III, we mine C code from open-source repositories. Al-

though an obvious solution to the synthesis of benchmarks, this approach is not common due to one fundamental shortcoming: it is difficult to compile code downloaded from repositories automatically, due to program dependencies. In the words of Cummins et al. [14]: "*preparing each of the thousands of open source projects to be directly applicable for learning compiler heuristics would be an insurmountable task.*" In this paper, we show how to compile these codes without human intervention. Key to the success of this endeavor is *type reconstruction*. We use PsycheC, a type inference engine for C [28], to fill up all the missing dependencies of code mined from the internet. This combination of code crawler and type reconstruction yields a collection of compilable programs that is practically unbounded.

Summary of Results. We call ANGHABENCH the collection of C benchmarks that we synthesize. Every benchmark is compilable, albeit non-executable. ANGHABENCH can be parsed by any C analyzer, can be converted into intermediate representations such as LLVM IR and gcc Gimple, and can be translated into object files. ANGHABENCH supports compiler tuning for code size reduction, and lets researchers study properties of real-world programs via static code analyses. Such possibilities are summarized in the following list of contributions:

- **Reconstructor**: Section III-A describes the infrastructure that we use to obtain compilable C programs from public repositories. This combination of web crawler and type inference engine is able to produce one million compilable C functions in about one week, including the time to download files and reconstruct types.
- **Distribution**: We provide a public collection of over one million compilable C files, organized as single-function and multi-function benchmarks. As we explain in Section III-B, this collection is browsable—search being guided by a vector of features that we extract from the LLVM representation [29] of each program.
- **Applications**: Section IV-B shows that ANGHABENCH predicts well the behavior of compiler optimizations. In Section IV-D, we use ANGHABENCH to train YACOS, a framework implemented by Filho *et al.* [6], [30] to find good optimization sequences for LLVM. Considering code size as the objective functions, we show that ANGHABENCH yields a training set 45.33% and 36.77% more effective than programs generated by CSMITH [31] and LDRGEN [32].
- **Optimization**: We have used ANGHABENCH to produce a code reduction tool, ANGHAZ, that improves clang-Oz by 11.1% on average. To compile an unseen program P_u, ANGHAZ searches a database of optimized functions for the program P_k the closest to P_u, given a well-known distance metric [5]. As discussed in Section IV-E, by applying onto P_u optimizations known to be effective on P_k, ANGHAZ can reduce codes impervious to even Rocha et al.'s state-of-the-art approach published in CGO19's distinguished paper [33].

c) Why compilation matters: We call "compilable" a file known in the C specification as a *translation unit* [34, Sec-5.1.1.1]. From a translation unit a C compiler can produce an object file. There are analyses that can be performed on the source code of programs, without the need to produce compilable code [35], [36], [37]. However, all the analyses and applications that we present in Section IV require compilable code, because these techniques either run on LLVM bytecodes or on machine code. In Section IV-F we show that type inference is essential to give us a large quantity of compilable benchmarks. Without type inference, we cannot go from the program's abstract syntax tree to object code, due to missing dependencies.

d) The benefits of a large code base: The reader could think that it is simple to analyze any partial C function, even if not compilable, as long as it is syntactically valid. This statement is not true. The C grammar is not context free; hence, most parsers, including clang's and gcc's, require all the dependencies in place, otherwise, statements like `T*c` become ambiguous: is `T` a type, or the first operand of a multiplication? Other ambiguities exist [28]. Having this large and unambiguous code base opens up many opportunities to understand real-world C programs. We have performed some analysis to this end, which, for the sake of space, we discuss in an extended technical report that accompanies this paper [38].

II. OVERVIEW

A. Predictive Compilation

As mentioned in Section I, a predictive compiler relies on properties of known programs to approximate properties of unknown programs. The collection formed by all the known programs is called the *Training Set*. Predictions, in this context, consist in matching *static program properties*, also called *features*, with *compilation actions*[1]. The concept of static program property has been defined in previous work; however, because this is a central notion to this paper, we recall its definition, using the notation proposed by Pereira *et al.* [39]:

Definition 1 (Static Program Feature)**.** Given a program P, a *static program feature* $f(P)$ is any characteristic of P, with the following attributes:

- **Static**: $f(P)$ depends only on the syntax of P;
- **Consistent**: if $f(P) = x$ and $f(P) = x'$, then $x = x'$;
- **Available**: $f(P)$ can be computed in polynomial time.

An ordered sequence of program features determines a *feature vector*. The set of every possible feature vector gives us a *feature space*. Such space can be explored in many different ways. For instance, because it abides by Euclidean Laws, it is sound to define distance between vectors. Example 1 illustrates these concepts.

[1]Previous work on predictive compilation, such as Zanella *et al*'s [30], also talk about dynamic program features. These are properties observed during the execution of programs. In this paper, we focus entirely on static program properties, as the benchmarks that we produce are not meant to run.

Example 1. Figure 1 shows a three-dimentional feature space. The three features that form it are Number of Instructions, Number of Stores and Loop Depth, i.e., the depth level of the innermost natural loop in the program.

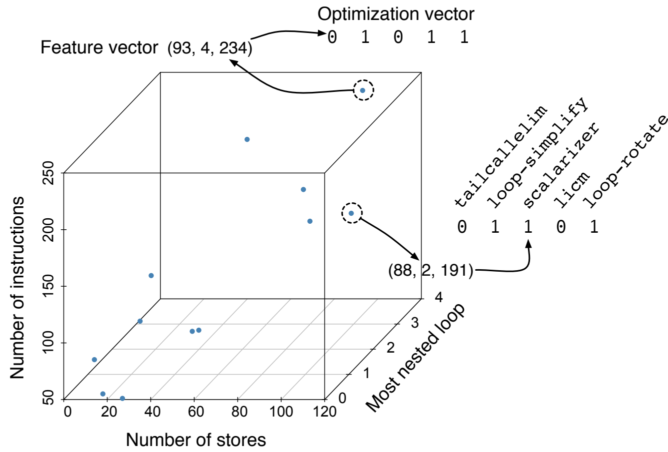

Fig. 1. Training a predictive compiler.

The *training phase* of a predictive compiler consists in a search, not necessarily exhaustive, for the most adequate compilation action for each program in the training set. The notion of "most adequate action" depends on two factors: (i) the objective function that guides the search; and (ii) the representation of the action. Typical objective functions include runtime, size and energy consumption. Common representations include tuples and lists of optimizations. In the former case, the order of application of an optimization is fixed—what varies is the occurrence or not of the optimization [6], [40]. In the latter, any permutation of a known universe of optimizations is acceptable [41], [42], [43].

Example 2. The property space seen in Figure 1 contains twelve programs, each one represented as a dot. The figure shows the feature vectors of two programs. The best tuple of optimizations, from a universe of five candidates, for each one of these two programs is also shown. A zero means that the optimization is inactive; a one means that it should be applied onto that program. Such tuples can be found using different heuristics, including exhaustive search. In this example, we assume that the objective function is size; thus, a tuple t_1 is better than a tuple t_2 when the optimizations in t_1 reduce code size more than the optimizations in t_2.

Once a compiler is trained, it can be used to optimize unknown programs. Optimizations, in this case, are based on approximations: the behavior observed in the training set is used to approximate the behavior of the unseen code. There are many ways to implement these approximations: neural networks, supporting vector machines, decision trees, etc. Example 3 uses one of such techniques: classification based on K-nearest neighbors [44], to perform predictions.

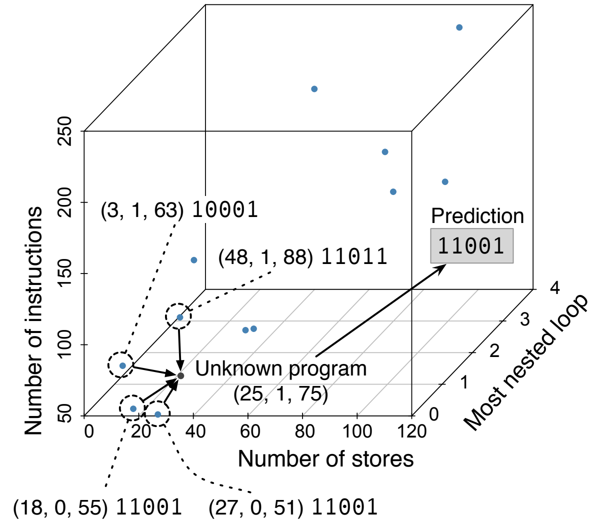

Fig. 2. Performing predictions.

Example 3. Figure 2 shows how the K-Nearest Neighbors algorithm can predict the best optimization tuple for a program. Given an unknown program with feature vector $(stores = 25, innermost = 1, instructions = 75)$, we find the four closest programs to this vector. The predictor activates the ith optimization if said optimization is active among the closest neighbors, and turns it off otherwise.

B. The Need for Benchmarks

If the training collection is small, then large chunks of the feature space will remain uncovered by the known codes. Compilers can still perform predictions; however, the information available during training might not approximate the behavior of unseen programs. To circumvent this problem, researchers often use synthetic benchmark suites. In this section, we analyze some of these approaches.

a) DeepSmith: One of the most successful benchmark generators in use today is DEEPSMITH [45], an evolution of Cummins *et al.*'s CLGEN [14]. DEEPSMITH has been able to produce realistic OpenCL programs. It is meant to be programming language agnostic; however, our attempts to use it towards generating C, instead of OpenCL, programs met with no success. Below, we narrate three of our experiences. In every case, we use compilable C programs drawn from a collection of half-a-million samples mined from open-source repositories as the initial training corpus:

- *Training set*: 30,000 randomly chosen C files. 107,264 candidate strings generated in 15 hours using a seed function signature with one argument. *Results:* Nine programs could be compiled. The largest LLVM bytecode had five instructions (Clang -O0).
- *Training set*: 30,000 randomly chosen C files. *Generation:* 131,760 candidate strings in 30 hours using a seed

signature with four arguments. *Results:* 1,178 programs could be successfully compiled. The largest program had six lines of code, and 36 LLVM instructions.
- *Training set*: the 10,000 largest C files in the available collection. *Generation:* 54,912 candidate strings generated in 10 hours using a seed function signature with four arguments. *Results:* Seventeen programs could be successfully compiled. The largest program had five lines of code, and 16 LLVM instructions.

b) Compiler Fuzzers: A compiler fuzzer produces random programs to uncover bugs in compilers. The most successful tool of this sort is CSMITH [31]. Programs generated by CSMITH have revealed hundreds of bugs in the LLVM infrastructure, and dozens in gcc's. LDRGEN, another tool of similar purposes, has also been effectively employed to find bugs in different compilers. Although tremendously successful as bug-finding resources, fuzzers are not meant to be used to generate training data for predictive compilers. Programs generated by fuzzers like CSMITH and LDRGEN tend to differ from real-world codes. Thus, properties inferred from them may not generalize to programs written by people. The next example supports this statement with empirical data.

Example 4. Figure 3 shows the relation between stores and loads in the 275 programs in the LLVM test suite. For each store, we find 3.35 load instructions. This analysis was performed in LLVM bytecodes compiled with -O0, but optimized with `mem2reg`. This optimization, `mem2reg`, has been used in this experiment to remove memory access operations related to stack-allocated variables. Without this optimization, binaries would contain too many loads and stores without a counterpart in the source code. Analyzing 10K programs generated by CSMITH, we find the inverse behavior: 0.47 loads per store. 10K programs produced by LDRGEN fare no better: they contain only one store instruction. The programs synthesized by DEEPSMITH (1,204 samples) approximate the ratio found in the actual benchmarks: for each store, we find 2.97 load instructions. Nevertheless, they are too small: the largest program contains only two store instructions.

III. THE ANGHABENCH COLLECTION

A. The Program Reconstruction Framework

We developed a completely automated process for generating compilable programs from open source projects. This infrastructure has three major components: (i) Repository Crawler; (ii) Function Extractor; and (iii) Type Inference Engine. Each of these steps is described in the rest of this section.

a) The Repository Crawler: The first stage in our framework consists in gathering the source-code from which we shall build benchmarks. Programs are mined from GitHub, via a web crawler. We filter out projects tagged as using the C programming language, then sort them by popularity (we use GitHub stars as a metric of popularity). The crawler traverses a prefix of this sorted list, whose length is determined by the user, cloning each of the repositories in order. The codebase of each repository is cleaned to remove files which are not C source or header files. This corpus of C programs is then provided as input to the Function Extractor.

b) The Function Extractor: The function extractor separates mined files into a collection of would-be programs, consisting of one C function per file. The extractor is a clang plugin, which runs after Clang's Abstract Syntax Tree building step. It traverses the program's AST, looking for function declarations. If a declaration is found and has a matching definition, we outline its implementation to a separate file. The plugin can run in two modes: it can either create a file for each function found, or one single file that aggregates all function definitions found within that input source file. Clang builds an AST for a program even if errors occur during compilation. However, unless dependencies can be solved, it cannot move from this point towards a final object file. Example 5 illustrates some issues that prevent compilation.

Example 5. Figure 4 shows a function extracted from the source code of the Tox peer-to-peer messaging application. This function (without the declarations in the grey box) is not compilable, namely because it calls another function whose declaration is unavailable in line 7, and contains references to an unknown type *BS_LIST* in lines 6, 7 and 12.

c) The Type Inference Engine: Once we have a large number of candidate programs, the next challenge is to make them compilable. To solve issues that prevent compilation, such as those seen in Example 5, we run each program through the PsycheC type inference engine [28]. PsycheC will fill the missing pieces within the candidate program, generating a version of its code that compiles.

Example 6. The grey box in Figure 4 shows the result of running the function `bs_list_find` through PsycheC. The resulting program has a function declaration for the missing function *find*, as well as a valid definition for the missing type *BS_LIST*. This is all the absent information that prevented compilation of the original code. Therefore, any C compiler can successfully compile this new version without errors.

On the Preservation of Information. PsycheC does not change the control flow graph of programs. However, changes may happen in the types of variables: fields present in structs and unions in the original program might be omitted in the reconstructed code, or might be declared with different types. Omissions happen if these fields are not used. Thus, the intermediate representations of the original and reconstructed programs are identical, except for the types of the variables.

B. The Code Distribution Framework

To distribute the programs assembled using the techniques seen in Section III-A, we have created a public website. Different benchmark suites can be downloaded from it. All these collections include only compilable codes. Compilation has been certified using gcc 7.5, LLVM v.6, v.8 and v10. Currently, we distribute the following suites:

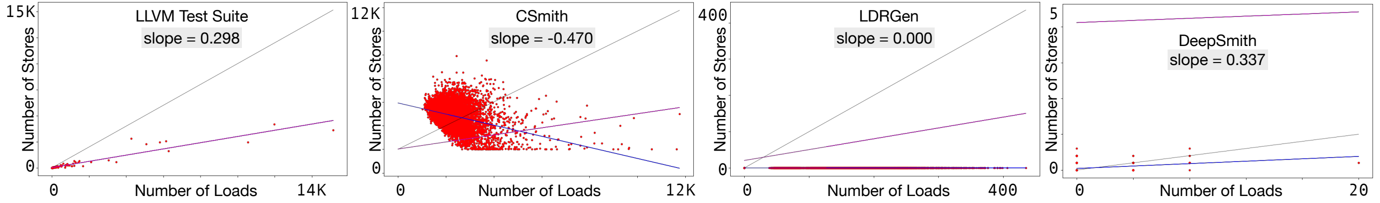

Fig. 3. A comparison between the number of stores and loads found in different benchmark collections. To ease visual comparison, each plot shows the line (in pink) produced for the programs in the LLVM test suite and the main diagonal (in grey), i.e., $X = Y$.

```
1  typedef int uint8_t;
2  struct TYPE_4__ { int* ids; };
3  typedef struct TYPE_4__ BS_LIST ;
4  int find (BS_LIST const*, int const*);
6  int bs_list_find
7    (const BS_LIST *list, const uint8_t *data) {
     int r = find(list, data);
8    //return only -1 and positive values
9    if (r < 0) {
10     return -1;
11   }
12   return list->ids[r];
13 }
```

Fig. 4. The code outside the grey area is an example of non-compilable candidate program extracted from the `toxcore` repository. The code in the grey area was introduced by PsycheC, to ensure compilation.

Collection	Quantity	Granularity	Mean	SD	Median
AnghaBench	1M	Functions	61.60	81.41	36
	530K	Functions	63.24	97.32	36
	10K	Functions	534.07	336.38	433
	15K	Whole files	266.64	419.79	119
CSmith	530K	Functions	5,844.67	5,876.67	3,933
	10K	Functions	20,190.90	3,649.04	19,161
LDRGen	530K	Functions	1,950.54	1,216.82	2,007
	10K	Functions	4,753.50	322.65	4,668
DeepSmith	1K	Functions	13.00	2.98	12
LLVM+SPEC06	288	Whole files	6,737.35	41,262.08	584

Fig. 5. Instructions per benchmarks in the collections that we distribute. SD is Standard Deviation.

- The ANGHABENCH collection:
  - a set with 1M files containing single functions;
  - a set with 530K files containing single functions;
  - the 10K largest files from the above set;
  - 15K files containing multiple functions.
- The LLVM test suite: 275 programs.
- The 10K largest programs among 530K programs generated with LDRGEN.
- The 10K largest programs among 530K programs generated with CSMITH.
- All the 1,204 programs that we have produced with DEEPSMITH (see Section II).

Figure 5 reports data about the size of the programs in the different collections. Program size is measured as the number of instructions of these programs in the LLVM intermediate representation. When converting programs to LLVM, we use the `mem2reg` pass, to move to virtual registers all the program variables that, otherwise, would be allocated on the stack.

a) A Protocol to Build Benchmarks: The largest ANGHABENCH collection contains 1,033,890 programs. For faster experiments, we also provide smaller suites. To build collections of N benchmarks, we follow the procedure below:

1) Let R be a list with the most popular git repositories, in number of stars, with a majority of files in C, in descending order, and let C be the collection of benchmarks.
2) While C has less than N files, we:
   a) Remove r, the current most popular repository from R;
   b) Add to C every function from r (see Section III-A).

b) The Code Search Engine: The public distribution contains a code search engine, which lets users retrieve the K closest program to a given code. Proximity is measured as the Euclidean Distance computed on the feature vectors introduced in Section II-A. We use LLVM to mine features from the intermediate representation of programs. Today, users can assemble vectors using features taken from a collection of 239 candidate program characteristics. We also provide three predefined feature vectors:

- LLVMSTS: 59 features produced by the LLVM's `--stats` flag applied on clang -O0.
- NUMERICAL FEATURES: 43 features taken from Filho *et al.* [6]. They project Namolaru *et al.* [5] features onto the LLVM IR.
- DEFAULT: A seven-features vector for fast searches, formed by (number of) *instructions*, *stores*, *loads*, *basic blocks*, *Control Flow edges*, *variables* and *variable uses*

Our similarity search does not relate programs based on semantic equivalence, à la Alon *et al.* [46]. Rather, close programs, in our context, are codes that tend to behave similarly when exposed to the same set of compiler optimizations. Therefore, it is assumed that the chosen feature vectors will relate programs by the effect that optimizations provoke on them. Notice, nevertheless, that while this is the objective, this result is not guaranteed to hold, given the statistical nature of every experiment described in this paper.

IV. EVALUATION

This section investigates the six research questions enumerated below. Further studies are available in the extended version of this work [38]:

RQA What is the mining rate of the infrastructure described in Section III-A?
RQB Can ANGHABENCH better predict the impact of compiler optimizations on programs?
RQC Can ANGHABENCH better approximate the properties of human-written benchmarks than code produced by other program synthesizers?
RQD How does ANGHABENCH compare to other synthetic benchmarks as training data for a predictive compiler?
RQE How does ANGHAZ compare to the state-of-the-art binary reducer of Rocha *et al* [33]?
RQF Can we build a collection similar to ANGHABENCH without the support of type inference?

Ground Truth. RQB presupposes the existence of a *ground truth*, that is, a collection of "typical" real-world benchmarks. Different benchmarks have been used at different times and places throughout the still short history of compilers. Therefore, finding a universally acceptable ground truth is an endeavor of improbable success. In this paper, we settle for a collection of 288 programs, which includes every benchmark available in the LLVM test collection (275 programs), plus the programs in the SPEC CINT CPU2006 suite (13 programs). In all, this collection gives us 1,450,035 lines of code, spread across 31,366 functions from 2,315 files.

A. RQA: Mining Throughput

This section investigates the rate in which the infrastructure from Section III-A produces valid benchmarks. A benchmark is considered valid when we can use both clang and gcc to convert it to an object file.

Methodology: We set up our framework to collect and reconstruct code, until a threshold of 1,000,000 compilable programs had been reached. The metric used for ranking repositories by popularity was GitHub's star feature. We executed the extraction-reconstruction process in parallel on an 8-core Intel i7-3770, with 16 GBs of RAM, running Ubuntu 16.04. We set a maximum execution timeout of 5 seconds for the type inference's constraint-solving step, as its unification algorithm has a potentially exponential worst-case performance [28]. We were concerned with answering two questions about our framework's performance:

- How long does it take, on average, to generate a compilable program?
- What is the success rate of the program reconstructor?

Discussion: To reach the threshold of 1,000,000 programs, our framework collected code from 148 repositories. It produced 1,044,023 compilable programs in 145 hours. This gives us an average rate of one benchmark per 0.5 seconds. In total, 1,882,687 candidate functions were extracted. Thus, the success rate for the reconstruction process was approximately 55.5%. Only 3,666 reconstructions failed due to the timeout. The most common reasons for failures were unpreprocessed macros that were not syntactically valid in C.

B. RQB: Predicting Compiler Behavior

The goal of this section is to support the thesis that ANGHABENCH predicts more accurately the behavior of compiler optimizations than other synthetic benchmarks suites.

Methodology: We shall investigate the code size reduction obtained by two different optimization levels of clang: -O1 and -O3, when applied on different benchmark collections. The choice for these two optimization levels is arbitrary; the ANGHABENCH distribution contains the same study for other optimization levels, and results are similar. We use the *Mean Square Prediction Error* (MSPE) as a measure of accuracy, defined as (predicted value − observed value)^2. To carry out predictions, we fit a linear model M in a synthetic benchmark, relating the size of programs when compiled with clang -O0 and clang -O1 (or -O3). M is then used to predict program size in the ground-truth collection.

Discussion: Figure 6 summarizes our findings. Each figure shows a main diagonal, and the regression line. Because optimizations tend to remove instructions, the regression line is always under the main diagonal. We give the slope of the regression line (`Slp`), and a measure of accuracy (`Err`). `Err` denotes the ratio between the MSPE of a given collection (ANGHABENCH, CSMITH, LDRGEN, DEEPSMITH) and the MSPE of the ground-truth. We compute `Err` as follows:

1) Let ℓ be the regression line that we fit into a given synthetic collection.
2) Let ℓ_g be the regression line that we fit into the ground-truth collection.
3) Let m_g be the MSPE that we obtain using ℓ_g on the ground-truth collection (this is the standard definition of MSPE).
4) Let m be the MSPE that we obtain when using ℓ also on the ground-truth collection.
5) We let $\texttt{Err} = m/m_g$.

The lower the value of `Err`, the better the predictor used to compute it. We report the error for the second largest collection in ANGHABENCH with 530K single functions (the collection with 1M samples makes this experiment too slow). Figure 6 shows that ANGHABENCH's error is one order of magnitude smaller than errors produced by the other collections. Programs from the CSMITH and from the LDRGEN collections are easy to optimize. They are made to execute without undefined behavior. To avoid undefinedness, they contain hardcoded inputs. The excess of constants leads to code that is easy to simplify. The programs generated by DEEPSMITH are also easy to optimize, although they do not contain hard-coded inputs (they are not meant to run). In this case, optimization opportunities come from an excess of dead-code.

C. RQC: Code Similarity

This paper defends the thesis that ANGHABENCH approximates more closely the properties of real-world code than other synthetic program sets. This section provides evidence that such is the case. To this end, we shall rely on the measure of "distance" between programs, which we discuss below.

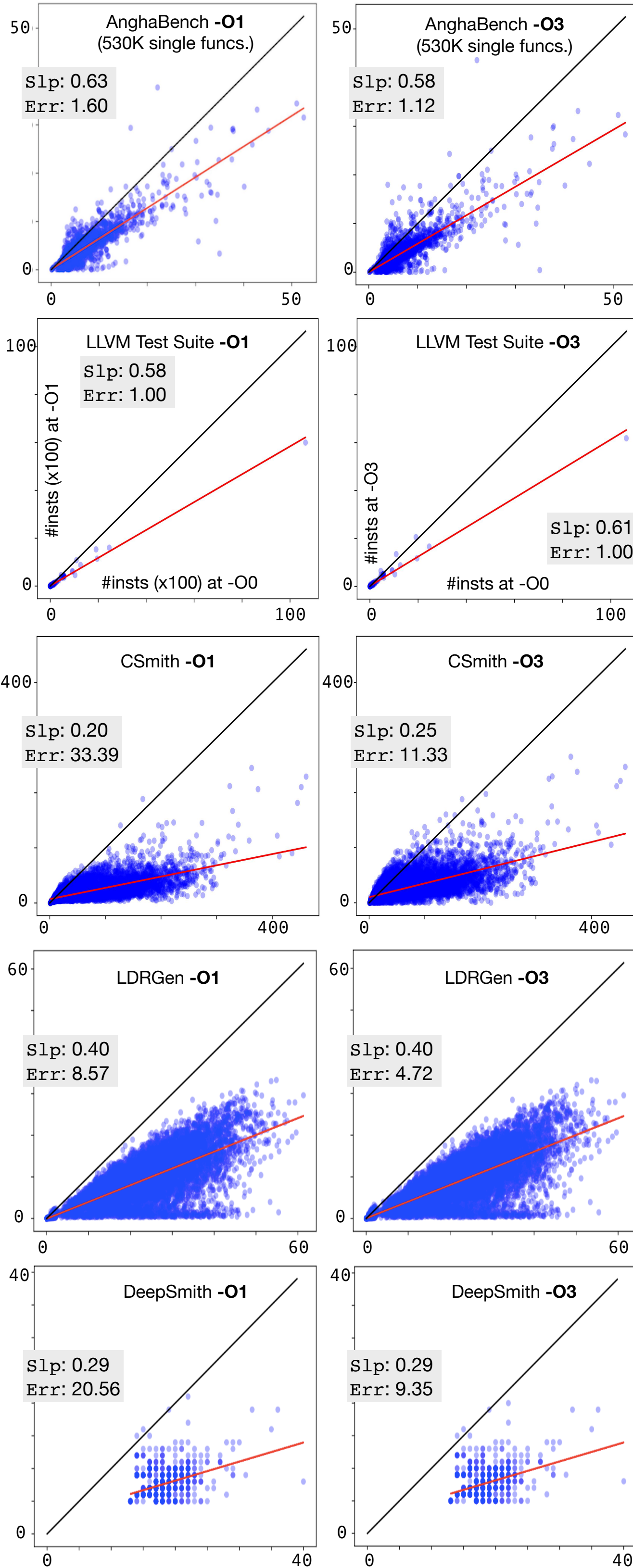

Fig. 6. The effect of compiler optimizations on the size of programs. Instructions are measured in hundreds.

Methodology: There exists a rich assortment of functions to measure the distance between programs [47]. We have adopted two of them: the Euclidean distance on numerical feature vectors, and the MCoeff relation between two programs proposed by Filho *et al.* [6]. The latter is a metric that measures how similarly two programs respond to the same sequence of compiler optimizations. Our choice is purely pragmatic: the infrastructure described in Section III-B already simplifies the computation of these two metrics. Furthermore, these two functions meet the following properties, assuming that p_1 and p_2 are programs: (i) $d(p_1, p_2) \geq 0$, (ii) $d(p_1, p_2) = d(p_2, p_1)$, and (iii) $d(p_1, p_2) \leq d(p_1, p_3) + d(p_3, p_2)$, for any $p_3 \notin \{p_1, p_2\}$. Finally, although there exist similarity metrics that are likely to be more expressive than those that we chose, e.g., à la DEEPSIM [48], they tend to be more costly to compute.

Discussion: Figure 7 shows the distance of each one of the 288 programs in the ground-truth to the different synthetic collections. The distance of a program p_g in the ground-truth collection to a collection C of synthetic benchmarks is given by $d(p_g, p_c)$, where p_c is the program in C that is the closest to p_g, and d is either the Euclidean distance on numerical feature vectors, or MCoeff.

The average Euclidean distance from the ground-truth collection to ANGHABENCH is 6.7x shorter than to the CSMITH (available in our public distribution) collection, and 33.3x shorter than to the LDRGEN (available in our public distribution—See Sec. III-B) collection. This difference is smaller once we consider MCoeff, but it is still noticeable. ANGHABENCH is approximately 21% and 29% closer to the ground-truth than the CSMITH collection and the LDRGEN collection, respectively.

Diversity. We use the notions of distance seen in this section to argue that ANGHABENCH is more diverse than the other synthetic collections that we use. The data in Figure 8 supports this statement. For each point $(N_b, K)_{(c,d)}$ in Figure 8, we assume that c is either ANGHABENCH, CSMITH, or LDRGEN, and d is a distance function, e.g., numerical features, or MCoeff. In this case, N_b is the number of benchmarks from collection c that wins as one of the K closest programs to some benchmark in the ground-truth collection. Thus, a very homogeneous collection c would have all the programs with the same features; leading to $(N_b, K)_{(c,d)} = K$ always. A very diverse collection, in turn, would give us $(N_b, K)_{(c,d)} = N_g \times K$, where N_g is the number of benchmarks in the ground-truth set. Therefore, according to these definitions, the larger $(N_b, K)_{(c,d)}$, the more heterogenous is collection c, and the better it covers the feature space.

D. RQD: Predictive Compilation

This section provides evidence that ANGHABENCH yields better training sets than other benchmark generators. To this end, we have used different synthetic collections of benchmarks to train YACOS, the predictive compiler implemented by Filho *et al.* [6], [30]. YACOS uses a heuristic based on Kennedy *et al.* [49]'s particle swarm optimization (PSO) to find good optimization sequences.

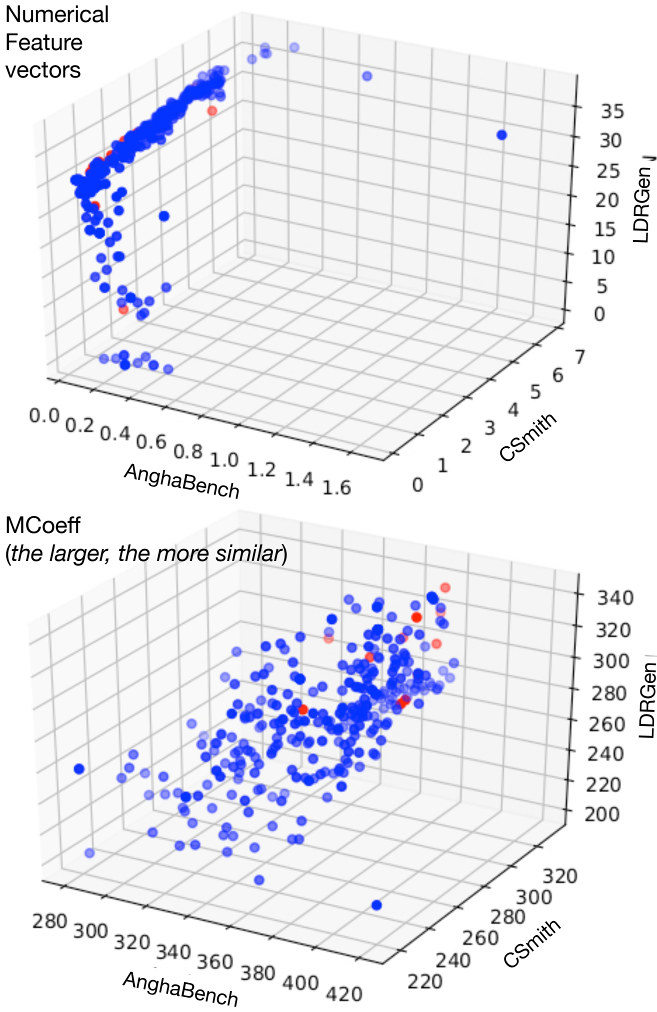

Fig. 7. Distance between the ground-truth and different synthetic benchmarks. Each collection is formed by the 10K largest programs out of a pool of 530K candidates. Red dots are programs from the SPEC CPU2006 suite.

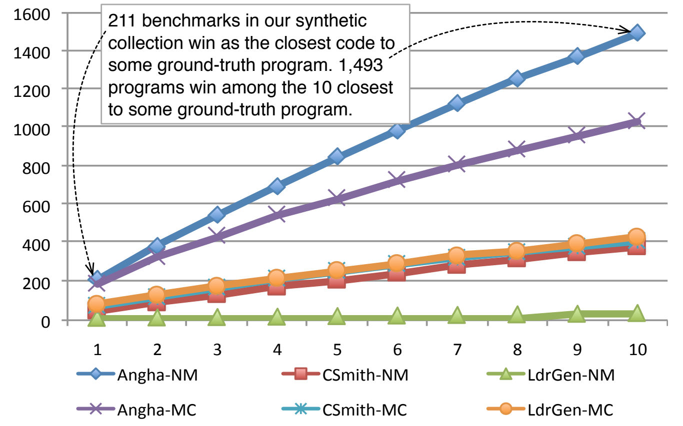

Fig. 8. Number of programs that win for K-nearest of some program in the ground-truth. The X-axis is K.

Methodology - Benchmarks: We have trained YACoS using the three collections mentioned in Section III-B, each with 10K files: ANGHABENCH, CSMITH and LDRGEN. For the last two collections, we produced 530K files, and took the 10K largest. We adopted this expedient for fairness, as the 10K programs from ANGHABENCH were selected from a collection of 530K benchmarks. Programs in the CSMITH collection are larger; hence, training based on them takes much longer: on average, three hours per file. The other two collections yield faster training time: on average 20 minutes per file. In total, training took 87 days, using 16 cores running at 3.40GHz.

Methodology - Training: Training consists in associating each benchmark in the training set with 100 sequences of optimizations—each sequence with 60 optimizations. These optimizations come from a set of 83 passes available in LLVM. Searching the feature space, in this setting, is the problem of associating with the feature vector of a program P the best list of optimizations for P. YACoS's PSO is parameterized with an initial population of 100 particles, which evolves for 10 generations. Nevertheless, the exact implementation of this heuristic is immaterial to the understanding of this paper—it suffices to know that its quality varies with the training set.

Methodology - Prediction: Once training is complete, YACoS uses similarity search (i.e., KNN with K=1) to find the known program P_k in the training set that is the closest to a given unknown program P_u. We have used the two measures of distance seen in Section IV-C: Euclidean distance applied on numerical feature vectors, and MCoeff. We let S_k be the list of optimizations associated with P_k. From S_k, YACoS can build different list of optimizations to be applied onto P_u following four strategies:

- **Elite**: let $S_e \subseteq S_k$ be a subset of S_k ($|S_k| = 100$) formed by the lists that improve on clang -Oz. We apply on P_u every sequence in S_e, and keep the best result. If S_e is empty, this strategy has no effect on P_u.
- **J**X, $X \in \{1, 10, 100\}$:] we apply on P_u **J**ust the X best sequences in S_k, and keep the best result. The sequences in S_k are ordered by their effect on P_k. The most effective optimization comes first.

Discussion: For validation, we use the ground-truth collection mentioned in Section IV-C. Figures 9 and 10 summarize the results of this experiment. A *winning strategy* is the pair in $\{\text{Elite}, \text{J1}, \text{J10}, \text{J100}\} \times \{\text{ANGHABENCH}, \text{CSMITH}, \text{LDRGEN}\}$ that yields the smallest bytecodes when applied to the validation set. We omit results involving benchmarks produced by DEEPSMITH. Due to their simplicity, their feature vectors contain mostly zeros. Boldface fonts mark winners considering minimum, maximum, mean and median code reduction. ANGHABENCH wins in most cases. When using the Euclidean Distance with the Elite choice, ANGHABENCH reduces code by 10.6% on average. If we use MCoeff, gains are higher: 11.1%. These results were not obtained in small programs: the ground-truth used as validation contains the 13 integer programs from SPEC CPU2006.

The gray cells in Figures 9 and 10 contain results for

	AnghaBench				CSmith				LDRGen			
	Elite	J1	J10	J100	Elite	J1	J10	J100	Elite	J1	J10	J100
min	-46	-86	-83	-46	**-15**	**-61**	**-61**	-33	-121	-145	-103	-33
avg	**10.6**	2.1	**8.6**	10.7	0.1	**3.9**	8.3	10.6	4.4	-7.3	6.5	10.7
med	**8.4**	3.5	7.4	8.4	0.0	**3.9**	**7.6**	**8.5**	3.5	-4.9	4.8	8.3
max	**41.2**	33.2	**36.4**	41.2	12.0	**34.2**	35.3	39.6	32.6	16.6	32.6	**44.4**
min	0.0	0.1	0.0	0.0	**2.9**	0.0	0.0	0.3	0.3	**0.8**	0.0	**0.6**
avg	11.1	**8.4**	**9.7**	11.0	7.8	7.6	9.5	11.0	8.2	4.6	8.4	**11.1**
med	**8.5**	**6.5**	**7.9**	8.5	8.0	6.3	7.7	8.6	6.5	3.3	6.6	**9.0**
max	**41.2**	33.2	**36.4**	41.2	12.0	**34.2**	35.3	39.6	32.6	16.6	32.6	**44.4**
#P	**280**	195	**269**	282	12	**211**	267	282	199	39	250	280

Fig. 9. Percentage of code reduction, measured over number of LLVM bytecodes, produced from the ground truth using YACOS parameterized with the Euclidean distance. **#P** reports the number of programs in which we have observed positive results over clang -Oz. Averages are geometric mean.

programs in which we could find a list of optimization better than clang -Oz. If we consider average values for these programs, then the difference between ANGHABENCH and the other collections is less apparent, as we are counting only positive results. However, ANGHABENCH is able to find non-trivial lists of optimization for more programs than the other synthetic collections. For instance, using the Elite strategy with Euclidean distance (Fig. 9), YACOS trained with ANGHABENCH was able to reduce the size of 280 programs (compared with clang -Oz), vs only 12 if we train YACOS with CSMITH, and 199, if we train YACOS with LDRGEN.

	AnghaBench				CSmith				LDRGen			
	Elite	J1	J10	J100	Elite	J1	J10	J100	Elite	J1	J10	J100
min	-38	**-98**	-61	**-38**	**-21**	-148	**-59**	-47	-55	-176	-70	-45
avg	**11.1**	**4.8**	**9.6**	**11.1**	0.7	-2.5	6.5	10.0	9.7	0.4	8.5	10.7
med	**9.3**	**4.8**	**8.1**	**9.3**	0.0	-1.2	4.5	8.2	7.8	3.2	6.8	8.6
max	**41.2**	**41.2**	**41.2**	**41.2**	31.4	21.5	38.0	38.5	37.4	37.4	37.4	37.4
min	**0.8**	0.0	**0.3**	**0.8**	0.2	0.1	0.1	0.2	0.1	**0.2**	0.1	0.1
avg	**11.5**	**8.3**	**10.6**	**11.4**	10.5	5.6	7.9	10.6	10.8	7.6	9.5	11.0
med	**9.5**	**6.4**	**8.7**	**9.5**	7.2	4.2	5.3	8.3	8.3	5.5	7.1	8.8
max	**41.2**	**41.2**	**41.2**	**41.2**	31.4	21.5	38.0	38.5	37.4	37.4	37.4	37.4
#P	**282**	**231**	**273**	**283**	24	112	251	277	269	192	271	282

Fig. 10. Percentage of reduction (same as Fig. 9) achieved on the ground truth using the MCoeff distance.

Using the ground-truth as the training set. We have also used the ground-truth (288 programs, including SPEC CINT CPUT2006) as the training set to predict good optimization sequences to itself. In this experiment, we have used a leave-one-out methodology with the MCoeff distance. Leave-one-out is applied per benchmark. The ground-truth is formed by the combination of 36 different benchmark collections. Thus, we use 35 suites as the training set, and one suite as the test set. On the J1 strategy, ANGHABENCH finds a better list of optimization than clang -Oz for 231 out of 288 programs. The ground-truth beats clang -Oz in only 30 programs. The average size reduction across all programs is 4.8% using ANGHABENCH (Figure 10) vs -14.2% using the ground-truth. On J10, ANGHABENCH outperforms clang -Oz in 273 programs, and the ground-truth in 109. On J100, they obtain statistically similar results: 283 vs 284 improvements. We emphasize that the ground-truth is much smaller than ANGHABENCH: we are comparing a training set of 10,000 samples with a training set with strictly less than 288.

E. RQE: Code Size Reduction

This section provides some perspective on the code size reduction achieved by a compiler trained with ANGHABENCH. To this end, we analyze the effects of this compiler on MIBENCH [50]. MIBENCH has been used by Rocha *et al.* [33] as a challenging case study. Rocha et al. have designed and implemented a technique to reduce code size by merging common sequences of instructions. Their *Function Merging by Sequence Alignment* (FMSA) approach excels when applied onto large code bases, as there are more opportunities for merging redundant code. However, their technique yields poor results when applied onto small programs—a natural consequence of a statistical lack of redundancies. Rocha et al. have used MIBENCH to demonstrate this last point.

Methodology: We compile MIBENCH with YACOS trained with the 10K largest programs from ANGHABENCH. Program similarity is measured with the Euclidean distance. Euclidean distance is used in this experiment because it is the default program metric used in YACOS. We use the subset of MIBENCH available in the LLVM test suite.

Discussion: Figure 11 reports the results that we have obtained after compiling MIBENCH with YACOS. The baseline is clang -Os. This is the same baseline adopted by Rocha et al. For reference, Figure 11 also reports, on top, the percentages of code size reduction observed by Rocha et al. Numbers refer to the size of the object file produced after compilation.

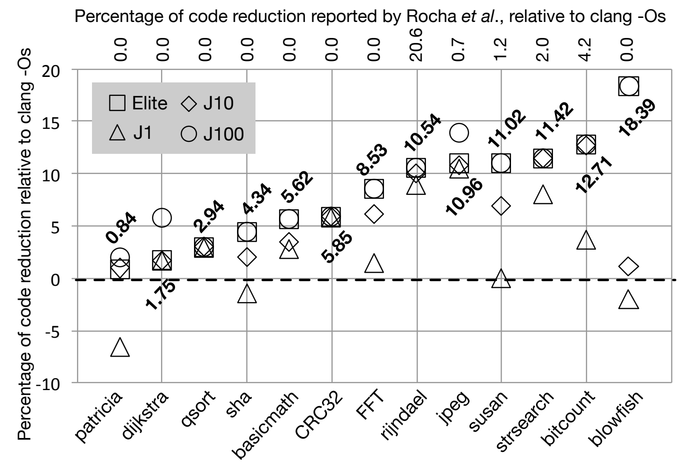

Fig. 11. Percentage of code size reduction achieved on MIBENCH, measured over size of object files, in bytes. Numbers on top are percentage of reduction reported by Rocha *et al.* [33]. Numbers at the bottom are percentage of reduction achieved by YACOS, using the Elite approach.

The amount of code size reduction obtained by YACOS tends to be higher than the reduction achieved by FMSA; however, the bad performance of FMSA on MIBENCH had been

noticed by Rocha *et al.* [33]. As we have mentioned, FMSA does better the larger the target code base, for in this case, there will be more redundancies to explore[2]. We emphasize that these numbers cannot be directly compared: they were not produced in the same empirical setup. Nevertheless, in at least eight benchmarks from MIBENCH, where Rocha et al. have reported no gains, we could observe reductions of 6.0% on average (geo-mean), compared to clang -Os. Another fact that this experiment highlights is that clang -Os and clang -Oz still leave much room for improvement. In benchmarks like `bitcount` and `strsearch` it is possible to find sequences of optimizations that are almost twice as efficient as clang -Os, and approximately 8% better than clang -Oz. Finally, it must be understood that FMSA is a compiler optimization, whereas our approach is rather a replacement to the LLVM pass manager. In other words, it would have been possible to include FMSA as one of the optimizations of YACOS's search strategy. We have not performed this new experiment simply because FMSA is not available by default in LLVM.

F. RQF: The Role of Type Inference

Provided that we can mine open-source repositories, and there are so many of them, one could expect that some of these programs naturally compile without type reconstruction. In this section, we show that such code base is worse than ANGHABENCH when used to train a predictive compiler.

1) The Need for Type Inference: One of the benchmark collections distributed in our website consists of 529,498 C functions and their respective LLVM bytecodes. This collection of over half-a-million compilable benchmarks has been produced out of 54,431 files taken from 79 open source repositories. Out of these files, we extracted 698,449 functions, sizes varying from one line to 45,263 lines of code (`Radare2`'s assembler). Thus, we produced an initial code base of 698,449 C files, each file containing a single function. We run PsycheC with a timeout of 30 seconds on this code base. PsycheC has been able to reconstruct dependencies of 529,498 functions; thus, ensuring their compilation. Compilation consists in the generation of an object file out of the function— a task performed with clang 6.0.1.

Out of the 698,449 functions, 31,935 were directly compilable as-is, that is, without PsycheC's inference. To perform automatic compilation, we invoke clang on a preprocessed C file containing an individual function extracted as-is. Hence, without type inference, we could ensure compilation of 4.6% of the programs. With type inference, we could ensure compilation of 75,8% of all the programs. Failures to reconstruct types were mostly due to macros that were not syntactically valid in C without preprocessing.

[2]Rocha *et al.* have published a new code size reduction technique that extends sequence alignment to SSA-form programs [51]. We believe that their new algorithm, SalSSA, is the most effective size reduction technique in use today. When applied onto MIBENCH, SalSSA achieves a geo-mean reduction of 1.4% to 1.6%; twice as much as FMSA. However, a direct comparison with our work is not possible, for they reported results only for ARM.

2) On the Consequence of using Small Functions: As we have mentioned, we can compile automatically less than 5% of the functions that we download, even considering all the dependencies in the C files where these functions exist. Nevertheless, given that we can download millions of functions, 5% is already enough to give us a non-negligible number of benchmarks. However, these compilable functions tend to be very small. The median number of LLVM bytecodes is seven (in contrast to 36, using type inference). Said functions are unlikely to contain features such as arrays of structs, type casts, recursive types, double pointer dereferences, etc. It is typical that developers separate definitions from implementations into different files in the C ecosystem. It suffices to have one missing file or the wrong version of one library, and the entire program will fail to compile.

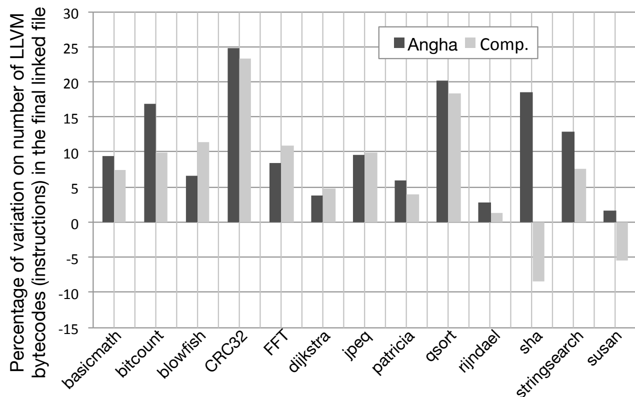

Fig. 12. Performance of YACOS (with the Elite choice) on the MIBENCH programs. We use two training sets: ANGHABENCH (10K largest functions) and the functions that we can compile without type inference. Percentage of reduction is measured with regards to clang -Oz on the number of LLVM bytecodes in optimized programs.

Due to their small size, the naturally compilable functions are not effective to tune a compiler. Figure 12 demonstrates this fact with data. The figure compares YACOS, when trained with our collection of half-a-million functions (from which we selected the 10K largest samples), or with the collection of 31.9K directly compilable functions (from which, again, we selected the 10K largest functions). The small size of the latter group prevents it from being an effective training set. For instance, using these functions, YACOS reduces the size of MiBench's `bitcount` by 10%, whereas using ANGHABENCH, it achieves 16.9%. In `susan`, the naturally compilable functions lead to an increase of code size (5.4%), whereas ANGHABENCH reduces it by 1.7%. Although there are benchmarks in MIBENCH where the naturally compilable functions lead to more compression, these gains are close to those obtained by ANGHABENCH, and seldom occur.

V. RELATED WORK

This paper deals with synthesis of benchmarks and code size reduction. Because the former has been discussed in Section II,

we now focus on the latter. Yet, we start our discussion mentioning some work on the synthesis of benchmarks.

Synthesis of Benchmarks. The creation of synthetic benchmarks has become a frequent focus of research. Random program generators, such as CSMITH [31], LDRGEN [32] and Orange3 [52], [53] have been successfully used to produce C programs for stress-testing compilers, often finding correctness bugs in industrial implementations [54]. Although conceived to find bugs, these tools have also been used to improve the quality of the optimized code emitted by mainstream C compilers [55], [56]. Nevertheless, such tools, given their goals, are not designed to produce realistic code.

Recent effort to create human-like C programs has leveraged Deep Learning techniques to generate code similar to real-world examples. CLGEN [14] uses this approach to generate OpenCL kernels, while DEEPSMITH [45] generalizes this technique to other languages. Nonetheless, we have found that DEEPSMITH has trouble synthesizing non-trivial C programs when trained with corpora of open source projects, as we have explained in Section II-B. We could not use it to train YACOS, in Section IV-D, because the feature vectors of its programs contained mostly zeros. Following a different approach, Richards *et al.* have produced realistic JavaScript benchmarks, out of monitored browser sections [57]. Type reconstruction is not an issue in this scenario, because JavaScript is dynamically typed. One shortcoming of this modus operandi is scalability, because it is not fully automatic. Richards' technique still requires users to create a browsing section, which will then be instrumented.

Code Size Reduction. Several compiler transformations have code reduction as either the main goal or a desirable consequence [33], [51], [58], [59], [60], [61], [62]. Often, such techniques involve *Code Factoring*, the identification of redundant code within the program. Code motion techniques search for identical instructions and merge them to avoid redundancy [62], [63], [64], [63]. *Function Merging* is the other main category in the field. This optimization finds functions that are semantically similar or equivalent, and generates new functions to replace them [65], [66], [67]. It is possible to merge even functions that are slightly different. When functions that meet a similarity threshold are found, a new procedure is created. The new function contains additional control-flow to choose between which original implementation should be executed [33], [51], [68].

In spite of program compression being an old problem, effective and elegant code size reduction techniques have been discovered as recently as the current year [51]. Nevertheless, the contributions of this paper are orthogonal to these techniques, because we do not propose new optimizations. Rather, our synthetic benchmarks can help a compiler identify when each of these optimizations might be profitable. To support this observation, in Section IV-E we showed how to augment a C compiler with knowledge extracted from our benchmarks.

Autotuning for Binary Size Reduction. Code size has been a common objective function of predictive compilers. The earliest works in this direction used genetic algorithms to continually improve the size of the code generated for target applications [42], [69], [70]. These early approaches evolve a sequence of optimizations for each individual program; thus, search runs until convergence for each program being optimized. The technique described in section IV-E, on the other hand, can simply find the program in the training set that better approximates the target application. Therefore, once the predictive model has been trained, the impact on compilation time is minimal.

VI. CONCLUSION

This paper has presented a framework to produce compilable C programs out of open-source repositories, which we have used to generate more than one million benchmarks. Compilation is ensured via type inference. Different applications of these benchmarks have been shown, with emphasis on predictive size reduction. In this regard, we showed that in benchmarks like `bitcount` and `blowfish`, from MIBENCH, it is possible to find sequences of optimizations that are about 15% better than clang -Os, and approximately 8% better than clang -Oz.

a) License: Although the process of building the compilable benchmarks is automatic, adding the licenses to the files is not. To fulfill this task, we need to find the license(s) used in the repository, which is not necessarily present in the program's source code. We have preserved the original licenses of a subset of 128,411 files from the following repositories: `FFmpeg`, `DeepMind`, `openssl`, `SoftEtherVPN`, `libgit2`, `php-src`, `radare2`, `darwin-xnu`, `mongoose`, `reactos`, `git`, `nodemcu-firmware`, `redis`, `h2o`, and `obs-studio`. These repositories give us a total of 38.9K C files. These files have been organized as an external LLVM test suite.

b) Further uses of ANGHABENCH*:* In addition to the applications seen in this paper, we are aware of a few other uses of ANGHABENCH. For instance, ANGHABENCH has been used to: (i) stress-test two C-to-Verilog compilers: LEGUP and VIVADO; (ii) test routing algorithms that convert C programs to FPGA circuits; (iii) fine-tune register assignment heuristics; (iv) compare the speed of C parsers; (v) check the effectiveness of a termination checker, the ULTIMATE Automizer; and (vi) scale up program synthesis, following an idea proposed by Bornholt and Torlak [71].

c) Software: ANGHABENCH is publicly available at http://cuda.dcc.ufmg.br/angha. In addition to the benchmarks, this webpage contains links to the infrastructure used to build them, including crawler, function extractor and type inference engine [28].

ACKNOWLEDGEMENT

This work has been made possible by grants from different research agencies, namely CNPq, CAPES and FAPEMIG. We thank Luigi Soares and Augusto Noronha for reading a draft of this paper. We also thank the CGO reviewers, for all the time and expertise that they have put into our manuscript.

REFERENCES

[1] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *Comput. Surv.*, vol. 51, no. 5, pp. 96:1–96:42, 2018.

[2] H. Leather and C. Cummins, "Machine learning in compilers: Past, present and future," in *FDL*. Washington, DC, USA: IEEE, 2020.

[3] Z. Wang and M. F. P. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[4] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "COBAYN: Compiler autotuning framework using bayesian networks," *TACO*, vol. 13, no. 2, pp. 21:1–21:25, 2016. [Online]. Available: http://doi.acm.org/10.1145/2928270

[5] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in *CASES*. New York, NY, USA: ACM, 2010, pp. 197–206.

[6] J. F. Filho, L. G. A. Rodriguez, and A. F. da Silva, "Yet another intelligent code-generating system: A flexible and low-cost solution," *J. Comput. Sci. Technol.*, vol. 33, no. 5, pp. 940–965, 2018.

[7] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," in *ASPLOS*. New York, NY, USA: ACM, 2016, pp. 297–310.

[8] T. C. de Souza Xavier and A. F. da Silva, "Exploration of compiler optimization sequences using a hybrid approach," *Computing and Informatics*, vol. 37, no. 1, pp. 165–185, 2018.

[9] M. Novaes, V. Petrucci, A. Gamatié, and F. M. Q. a. Pereira, "Compiler-assisted adaptive program scheduling in big.little systems: Poster," in *PPoPP*. New York, NY, USA: ACM, 2019, pp. 429–430.

[10] J. K. V. Sreelatha, S. Balachandran, and R. Nasre, "CHOAMP: cost based hardware optimization for asymmetric multicore processors," *Trans. Multi-Scale Computing Systems*, vol. 4, no. 2, pp. 163–176, 2018.

[11] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *OSDI*. Berkeley, CA, USA: USENIX Association, 2008, pp. 177–192.

[12] R. Bunel, A. Desmaison, M. P. Kumar, P. H. S. Torr, and P. Kohli, "Learning to superoptimize programs," in *ICLR*. Toulon, France: OpenReview, 2017.

[13] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Commun. ACM*, vol. 59, no. 2, pp. 114–122, 2016.

[14] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *CGO*. Piscataway, NJ, USA: IEEE, 2017, pp. 86–99.

[15] G. Poesia, B. C. F. Guimarães, F. Ferracioli, and F. M. Q. Pereira, "Static placement of computation on heterogeneous devices," *PACMPL*, vol. 1, no. OOPSLA, pp. 50:1–50:28, 2017.

[16] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni, "Automatic construction of inlining heuristics using machine learning," in *CGO*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12.

[17] P. Zhao and J. N. Amaral, "To inline or not to inline? enhanced inlining decisions," in *LCPC*. Heidelberg, Germany: Springer, 2003, pp. 405–419.

[18] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[19] R. Bagrodia, R. Meyer, M. Takai, Y.-a. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: A parallel simulation environment for complex systems," *Computer*, vol. 31, no. 10, pp. 77–85, 1998. [Online]. Available: https://doi.org/10.1109/2.722293

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. Washington, DC, USA: IEEE, 2009, pp. 44–54.

[21] L.-N. Pouchet and T. Yuki, "PolyBench/C 4.2.1: The polyhedral C benchmark suite," 2018. [Online]. Available: http://polybench.sf.net

[22] G. Fursin and O. Temam, "Collective optimization: A practical collaborative approach," *Trans. Archit. Code Optim.*, vol. 7, no. 4, pp. 20:1–20:29, 2010.

[23] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: http://dx.doi.org/10.1177/109434209100500306

[24] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, "A black-box approach to energy-aware scheduling on integrated cpu-gpu systems," in *CGO*. New York, NY, USA: ACM, 2016, pp. 70–81.

[25] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *PPoPP*. New York, NY, USA: ACM, 2009, pp. 75–84.

[26] Z. Wang and M. F. O Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *PACT*. New York, NY, USA: ACM, 2010, pp. 307–318.

[27] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for opencl programs on CPU/GPU heterogeneous platforms," in *HiPC*. Los Alamitos, CA, USA: IEEE, 2014, pp. 1–10.

[28] L. T. C. Melo, R. G. Ribeiro, M. R. de Araújo, and F. M. Q. a. Pereira, "Inference of static semantics for incomplete c programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 29:1–29:28, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3158117

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. Washington, DC, USA: IEEE, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[30] A. F. Zanella, A. F. da Silva, and F. M. Q. ao Pereira, "YACOS: a complete infrastructure to the design and exploration of code optimization sequences," in *SBLP*. New York, NY, USA: ACM, 2020, pp. 56–63.

[31] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*. New York, NY, USA: ACM, 2011, pp. 283–294.

[32] G. Barany, "Liveness-driven random program generation," in *LOPSTR*. Heidelberg, Germany: Springer, 2017, pp. 112–127.

[33] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *CGO*. Piscataway, NJ, USA: IEEE Press, 2019, pp. 149–163.

[34] ISO-Standard, "ISO/IEC 9899:tc3 - committee draft of the C99 standard," 2011.

[35] R. Dyer, H. Rajan, and Y. Cai, *Language Features for Software Evolution and Aspect-Oriented Interfaces: An Exploratory Study*. Berlin, Heidelberg: Springer-Verlag, 2013, p. 148–183.

[36] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *ICSE*. New York, NY, USA: ACM, 2014, p. 779–790.

[37] J. Eyolfson and P. Lam, "How C++ developers use immutability declarations: An empirical study," in *ICSE*. Washington, DC, USA: IEEE, 2019, p. 362–372.

[38] A. F. da Silva, B. Kind, J. W. M. aes, J. Rocha, B. G. aes, and F. M. Q. ao Pereira, "Anghabench: a synthetic collection of benchmarks mined from open-source repositories," Universidade Federal de Minas Gerais, Tech. Rep. 01-2020, 2020.

[39] F. M. Q. a. Pereira, G. V. Leobas, and A. Gamatié, "Static prediction of silent stores," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 44:1–44:26, 2018.

[40] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost GCC: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[41] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 77–90. [Online]. Available: http://doi.acm.org/10.1145/781131.781141

[42] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '99. New York, NY, USA: ACM, 1999, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/314403.314414

[43] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305. [Online]. Available: http://dx.doi.org/10.1109/CGO.2006.37

[44] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Trans. Inf. Theor.*, vol. 13, no. 1, pp. 21–27, 2006.

[45] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *ISSTA*. New York, NY, USA: ACM, 2018, pp. 95–105.

[46] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[47] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *CSMR*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 45–54.

[48] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *ESEC/FSE*. New York, NY, USA: ACM, 2018, p. 141–151.

[49] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of the International Conference on Neural Networks*, vol. 4, Nov 1995, pp. 1942–1948 vol.4.

[50] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *WWC*. Washington, DC, USA: IEEE, 2001, pp. 3–14.

[51] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective function merging in the ssa form," in *PLDI*. New York, NY, USA: ACM, 2020, p. 854–868.

[52] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions," *IPSJ Trans. System LSI Design Methodology*, vol. 7, pp. 91–100, 2014.

[53] K. Nakamura and N. Ishiura, "Introducing loop statements in random testing of c compilers based on expected value calculation," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, 2015, pp. 226–227.

[54] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, 2020.

[55] A. Hashimoto and N. Ishiura, "Detecting arithmetic optimization opportunities for c compilers by randomly generated equivalent programs," *IPSJ Transactions on System LSI Design Methodology*, vol. 9, pp. 21–29, 2016.

[56] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 82–92. [Online]. Available: http://doi.acm.org/10.1145/3178372.3179521

[57] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of javascript benchmarks," *SIGPLAN Not.*, vol. 46, no. 10, p. 677–694, 2011.

[58] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," *SIGPLAN Not.*, vol. 29, no. 6, pp. 147–158, Jun. 1994. [Online]. Available: http://doi.acm.org/10.1145/773473.178256

[59] J. Cocke, "Global common subexpression elimination," *SIGPLAN Not.*, vol. 5, no. 7, pp. 20–24, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808480

[60] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco, "Code compression," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 358–365. [Online]. Available: http://doi.acm.org/10.1145/258915.258947

[61] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982. [Online]. Available: http://doi.acm.org/10.1145/357153.357155

[62] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 159–170. [Online]. Available: http://doi.acm.org/10.1145/178243.178257

[63] G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes, "Code factoring in gcc," in *Proceedings of the 2004 GCC Developers' Summit*, 2004, pp. 79–84.

[64] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-based procedural abstraction," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–270. [Online]. Available: https://doi.org/10.1109/CGO.2007.14

[65] S. Tallam, C. Coutant, I. L. Taylor, X. D. Li, and C. Demetriou, "Safe icf: Pointer safe and unwinding aware identical code folding in gold," in *GCC Developers Summit*, 2010. [Online]. Available: http://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=tallam.pdf

[66] M. Liska, "Optimizing large applications," *arXiv*, 2014.

[67] L. C. Infrastructure. (2019, nov) Mergefunctions pass, how it works. [Online]. Available: http://llvm.org/docs/MergeFunctions.html#mergefunctions-pass-how-it-works

[68] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/2597809.2597811

[69] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan, "Finding effective optimization phase sequences," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, ser. LCTES '03. New York, NY, USA: ACM, 2003, pp. 12–23. [Online]. Available: http://doi.acm.org/10.1145/780732.780735

[70] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, Sep. 2004, pp. 65–74.

[71] J. Bornholt and E. Torlak, "Scaling program synthesis by exploiting existing code," 2015.