Compilers Laboratory
lac.dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
UFMG

AnghaBench: a Synthetic Collection of Benchmarks Mined from Open-Source Repositories

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

Bruno Conde Kind, UFMG, Brazil - condekind@dcc.ufmg.br

José Wesley de Souza Magalhães, UFMG, Brazil - josewesleysouza@dcc.ufmg.br

Jerônimo Nunes Rocha, UFMG, Brazil - jeronimonunes@dcc.ufmg.br

Breno Campos Ferreira Guimarães, UFMG, Brazil - brenosfg@dcc.ufmg.br

Fernando Magno Quintão Pereira, UFMG, Brazil - fernando@dcc.ufmg.br


```
@techreport{Faustino20,
  title = {AnghaBench: a Synthetic Collection of Benchmarks
     Mined from Open-Source Repositories},
  author = {Anderson Faustino and Bruno Kind and Jos\'{e} Wesley
    Magalh\~{a}es and Jer\^{o}nimo Rocha and Breno Guimar\~{a}es
    and Fernando Magno Quint\~{a}o Pereira},
  year = {2020},
  institution = {Universidade Federal de Minas Gerais},
  number = {01-2020}
}
```


AnghaBench: a Synthetic Collection of Benchmarks Mined from Open-Source Repositories

ANDERSON FAUSTINO DA SILVA, UEM, Brazil
BRUNO CONDE KIND, UFMG, Brazil
JOSÉ WESLEY DE SOUZA MAGALHÃES, UFMG, Brazil
JERÔNIMO NUNES ROCHA, UFMG, Brazil
BRENO CAMPOS FERREIRA GUIMARÃES, UFMG, Brazil
FERNANDO M Q PEREIRA, UFMG, Brazil

A predictive compiler uses properties of a program to decide how to optimize it. In this scenario, the compiler is trained onto a collection of programs to derive a model which determines its actions in face of unknown codes. One of the greatest challenges of predictive compilation is how to find good training sets. Regardless of the programming language, the availability of human-made benchmarks is limited. In addition, 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 Angha, a virtually unlimited collection of real-world compilable C programs. To demonstrate the applicability of Angha, we show how it closely mimics properties of typical C benchmarks. From this observation, we use thousands of its programs to train predictive compilers, greatly outperforming results obtained with synthetic training sets.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Compilers**; *Software libraries and repositories*;

Additional Key Words and Phrases: Benchmark, Repository, Synthesis, Training

ACM Reference Format:
Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando M Q Pereira. 2020. AnghaBench: a Synthetic Collection of Benchmarks Mined from Open-Source Repositories. 1, 1 (May 2020), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The growing popularity of stochastic classification techniques is contributing to make compiler autotuning an effective approach to the generation of efficient programs [3, 60]. Autotuning is implemented as follows. A compiler is trained on a collection of programs, and uses results learned

Authors' addresses: Anderson Faustino da Silva, DIN, UEM, 5790 Colombo Avenue, Maringá, Paraná, 87020-900, Brazil, anderson@din.uem.br; Bruno Conde Kind, DCC, UFMG, Avenida Antônio Carlos, 6627, Belo Horizonte, Minas Gerais, 31.270-213, Brazil, condekind@dcc.ufmg.br; José Wesley de Souza Magalhães, DCC, UFMG, Avenida Antônio Carlos, 6627, Belo Horizonte, Minas Gerais, 31.270-213, Brazil, josewesleysouza@dcc.ufmg.br; Jerônimo Nunes Rocha, DCC, UFMG, Avenida Antônio Carlos, 6627, Belo Horizonte, Minas Gerais, 31.270-213, Brazil, jeronimonunes@dcc.ufmg.br; Breno Campos Ferreira Guimarães, DCC, UFMG, Avenida Antônio Carlos, 6627, Belo Horizonte, Minas Gerais, 31.270-213, Brazil, brenosfg@dcc.ufmg.br; Fernando M Q Pereira, DCC, UFMG, Avenida Antônio Carlos, 6627, Belo Horizonte, Minas Gerais, 31.270-213, Brazil, fernando@dcc.ufmg.br.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
© 2020 Tex format taken from Association for Computing Machinery.
/2020/5-ART $0.00 (Free of Charges)
https://doi.org/10.1145/nnnnnnn.nnnnnnn

during this training phase to derive a model which infers how to optimize unseen codes. During training, samples from the known collection of programs are compiled in many different ways, and best results, given some objective function such as runtime, size or energy consumption, are recorded. When faced with an unknown program P_u, the compiler uses the model to guide its action on which analyses and optimizations to apply. This modus operandi has been shown to be effective along different dimensions of code efficiency, such as runtime [4, 21, 26, 44, 48], energy consumption [46, 54], code size [8, 14, 52], hardware usage [20, 49], and the size-speed relation [53, 63], for instance.

The Problem Posed by the Lack of Benchmarks. A common shortcoming in this field, extensively discussed by Cummins *et al* [20], is the small size of typical training sets. To demonstrate this point, 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 [33] contains 12, SPEC CFP2006 [33] contains 17, Parsec [5] contains 13, Rodinia [15] contains 23, Polybench [50] contains 30, cBench [28] contains 30, and NPB v.1 [6] 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.*" [60]

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 [11, 54, 58, 59, 61]. 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.* [54] 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.

Our Contributions. In this paper, we bring forward a new technique to generate compilable benchmarks for the C programming language. As we explain in Section 3, our methodology is based on the automatic download of C code from open-source repositories. Although an obvious alternative to the synthesis of vast amounts of benchmarks, this approach is not common practice due to one fundamental shortcoming: it is difficult to compile code downloaded from repositories automatically, due to program dependences. In the words of Cummins *et al.* [20]: "*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 [41], to fill up all the missing dependences of code mined from the internet. This combination of code crawler and type reconstruction lets us build a collection of compilable programs that, for all practical purposes, is virtually unbounded.

Summary of Results. We call the collection of C benchmarks that we synthesize out of open-source repositories, Angha. This collection contains only compilable code; however, similarly to other synthetic benchmark generators[7, 9, 19, 20, 31], we do not guarantee the absence of undefined behavior when such programs run. Nevertheless, we show that the existence of this

very large collection of programs, plus the infrastructure to augment it at will is beneficial to the programming language community. The reader could think that it is simple to analyze any partial C function, even if it does not compile, 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 dependences in place, otherwise, statements like `T*c` become ambiguous: is `T` a type, or the first operand of a multiplication? Melo et al. [41] present other examples of ambiguities. ANGHA, in turn, 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. ANGHA 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 3.1 describes the infrastructure that we use to obtain compilable C programs out of open-source 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, extract functions and reconstruct missing types.

Distribution We currently provide a public collection of over half-a-million compilable C files, organized as single- function and multiple-function benchmarks. As we explain in Section 3.2, this universe can be browsed in different ways –search being enabled by analysis of the LLVM representation [38] of each program.

Analyses In Section 4.2 we apply static analyses on C programs to obtain a glimpse on properties that are representative of real-world code, such as uses-per-variables, loads-per-stores, instructions-per-branches, etc. Section 4.3 shows that ANGHA approximates these properties for human-written benchmarks substantially better than codes produced by synthesizers such as CSMITH [62], LDRGEN [9] and DeepSmith [19].

Applications In Section 4.5, we use ANGHA to train YACOS, a framework implemented by Filho et al. [26] to find good optimization sequences for LLVM. Considering code size as the objective functions, we show that ANGHA yields a training set 45.33% and 36.77% more effective than programs generated by CSMITH [62] and LDRGEN [9].

Optimization We have used ANGHA to produce a code reduction tool, ANGHAZ, that improves clang-Oz by 11.1% on average. ANGHAZ, subject of Section 4.6, augments clang with a database of known programs. To compile an unseen program P_u, it finds the program P_k the closest to P_u, using well-known metrics of program distance [44]. By applying onto P_u optimizations effective on P_k, ANGHAZ can reduce codes impervious to even Rocha et al. [51]'s state-of-the-art approach.

2 OVERVIEW

2.1 Predictive Compilation

As mentioned in Section 1, 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 *program properties*, also called *features*, with *compilation actions*. The concept of program property has been defined in previous work; however, because this is a central notion to this work, we recall its definition, using the notation proposed by Pereira et al. [47]:

Definition 2.1 (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 $f(P) = x$, then x is unique;
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 2.2 illustrates these concepts.

Example 2.2. Figure 1 shows a three-dimentional feature space. The three features that form it are Number of Instructions, Number of Stores and Loop Depth. The latter feature is determined by the depth level of the innermost natural loop in the program.

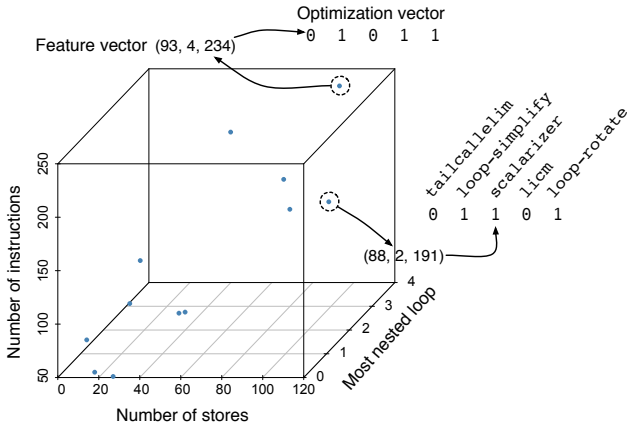

Fig. 1. Training a predictive compiler.

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 [26, 27]. In the latter, any permutation of a known universe of optimizations is acceptable [1, 17, 55].

Example 2.3. 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 2.4 uses one of such techniques: classification based on K-nearest neighbors [18], to perform predictions.

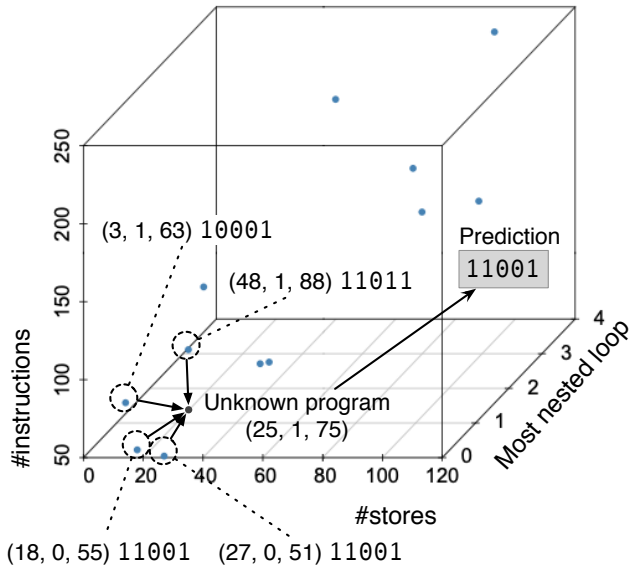

Fig. 2. Performing predictions.

Example 2.4. Figure 2 shows how the K-Nearest Neighbors algorithm can be used to 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 best compilation action for this vector activates the i^{th} optimization if said optimization is active among the closest neighbors, and turns it off otherwise.

2.2 The Need for Benchmarks

If training is performed with a small collection of benchmarks, then large chunks of the feature space will remain uncovered by the known codes. Compilers can still perform predictions, given the information made available during training; however, this information might not approximate the behavior of unseen programs, as Example 2.5 details.

Example 2.5. The twelve programs used in our previous examples leave a large portion of the feature space uncovered, which Figure 3 shows. When faced with an unknown program P, a compiler can still find its four closest neighbors, like in Example 2.4. However, these programs are too different from P to approximate its properties. Predictions performed in this case are unlikely to be accurate.

Example 2.6. One of the most well-established efforts in the field of predictive compilation for the C programming language is the Milepost GCC project [27]. Researchers involved in this project have assembled a training set of programs, and have extracted static features to represent each program. Training data is collected by compiling programs in the training set with varying sequences of optimizations, and recording how each sequence performs. Different machine learning models use the knowledge acquired during training to predict which sequence to use when optimizing an unseen program. Milepost GCC has been used to optimize for runtime and code size. As well-engineered as this project is, its main training set still consists of only 21 programs.

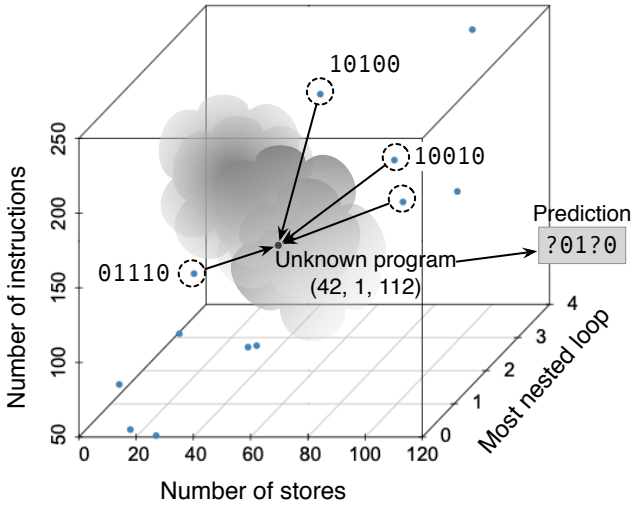

Fig. 3. Space not covered by programs in the training set.

2.3 Synthetic Program Generators

DeepSmith. There exist tools that generate synthetic programs in different programming languages. One of the most recent elements in this family is DeepSmith [19], an evolution of Cummins *et al.*'s CLGen [20]. DeepSmith has been shown to be able to produce impressively realistic OpenCL programs. It is meant to be programming language agnostic; however, our attempts to use it towards generating C 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, extracted from the program in Figure 4(a) had five instructions.
- *Training set*: 30,000 randomly chosen C files. *Generation:* 131,760 candidate strings generated in 30 hours using a seed function signature with four arguments. *Results:* 1,178 programs could be successfully compiled. The largest program, shown in Figure 4(b) had six lines of code, and yielded 36 instructions when converted to the LLVM representation.
- *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, shown in Figure 4(c) had five lines of code, and led to a program in LLVM assembly with 16 instructions.

```
void A(*a) {
  E(a, "mapping error_ %u e-chain");
  return;
}
```

(a)

```
void A(int*a, int*b, float c, char*d) {
  for (c = 0; c > 0; c++)
    for (c = 0; c < c; c++)
      c--;
  c--;
}
```

(b)

```
void A(int*a, int*b, float c, char*d) {
  "Thread : error 00000\n";
  int e, f, g, h;
  f = c;
}
```

(c)

Fig. 4. Largest programs produced by DeepSmith in our experimental setup.

We speculate that, when trained onto a relatively homogeneous corpus of programs, like the OpenCL kernels of the original publication [19, 20], DeepSmith can produce realistic benchmarks; however, when given a varied training collection, this Markov-based technique still struggles to produce good codes.

Compiler Fuzzers. A compiler fuzzer produces random programs to uncover bugs in compilers. The most successful tool of this sort is CSmith [62]. CSmith is easy to use, and very effective. 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-funding 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 2.7. As Figure 5 shows, the relation between stores and loads in the LLVM test suite, a collection formed by 275 benchmarks, is 0.298. In other words, for each store found in a typical program, we tend to find 3.35 load instructions[1] Analyzing the same metric into 10K programs generated by CSmith, we find the inverse behavior: for each store we have 0.47 loads. 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.

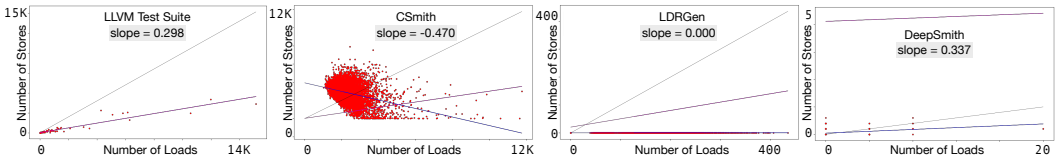

Fig. 5. A comparison between the number of stores and loads found in different benchmark collections. To ease visual comparison, each plot shows the line (in pink) produced for the programs in the LLVM test suite.

The programming language community has done impressive work towards the generation of benchmarks. However, example 2.7, plus the discussion in this section, indicate that there still remains a long way to tread until we can have benchmarks useful to train predictive compilers. In the next section, we introduce the collection that we have created, and explain the infrastructure used in this construction.

3 THE ANGHA COLLECTION

3.1 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. When used in combination, these three parts allow us to build, from raw C files available in the web, a virtually unbounded number of compilable benchmarks. The following paragraphs describe the goal of each step in greater detail.

[1]This analysis was performed in LLVM bytecodes compiled with -O0, but optimized with the `mem2reg` pass.

The Repository Crawler. The first stage in our framework consists in gathering the source-code from which we shall build benchmarks. To this end, we resort to the *largest host* [29] of open-source code in the world: GitHub. We built a simple web crawler that leverages GitHub's public API to download code from large open source C repositories. 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)[2].

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 next stage in the framework: the Function Extractor.

The Function Extractor. Once we have built a large body of C source files, the function extractor separates them into a collection of would-be programs, consisting of one C function per file. To perform this task, we use Clang, the C frontend of the LLVM Compilation Infrastructure. We built a plugin which runs after Clang's Abstract Syntax Tree building step. It traverses the program's AST, looking for function declaration nodes. 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 the input source file. Clang builds an AST for a program even if errors occur during compilation. However, unless dependences can be solved, it cannot move from this point towards a final object file. Example 3.1 illustrates some issues that prevent compilation.

```
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(const BS_LIST *list, const uint8_t *data) {
7    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. 6. 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.

Example 3.1. Figure 6 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.

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 3.1, we run each program through the PsycheC type inference engine [41]. PsycheC will fill the missing pieces within the candidate program, generating a version of its code that compiles.

Example 3.2. The grey box in Figure 6 shows the result of running the function `bs_list_find` through PsycheC. The resulting program has a function declaration for the missing function *find*,

[2]For reference, the top five repositories in this list are the Linux operating system kernel (https://github.com/torvalds/linux), the Netdata system monitor (https://github.com/netdata/netdata), the Redis distributed database (https://github.com/antirez/redis), the Git version control system (https://github.com/git/git) and the PHP interpreter reference implementation (https://github.com/php/php-src).

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.

There are two major caveats to this process. First, we stress that the programs we generate are *compilable*, but not necessarily *executable*. For instance, while the new program in Figure 6 contains a declaration for the missing function, it does not contain a definition for it. Thus, this program cannot be linked. In Section 4 we present different ways to benefit from this benchmark, even though it is not executable. Second, there is no guarantee that the resulting code that PsycheC generates is semantically equivalent to its original counterpart, as Example 3.3 illustrates. Thus, the benchmarks we generate are an approximation of the codes mined from the repositories, rather than equivalent to them.

Example 3.3. For example, the definition for the *BS_LIST* generated for the program in Figure 6 contains only a single field *ids*. The actual definition for this type in the original project is much more complex. However, since this is its only field used within the function, PsycheC generates a much simpler version of it.

3.2 The Code Distribution Framework

To distribute the programs assembled using the techniques seen in Section 3.1, 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 LLVM v.6, v.8 and v10. Currently, we distribute the following suites:

- The Angha collection:
  - a set with 530K files containing single functions;
  - the 10K largest files from the above set;
  - 15K files containing multiple functions.
- Actual benchmarks: 275 programs taken from the LLVM Test-Suite.
- 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 2.3).

Figure 7 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 in stack.

	Mean	SD	Median
Mystery 530K functions	63.24	97.32	36
Mystery 10K functions	534.07	336.38	433
Mystery 15K Files	266.64	419.79	119
CSmith 530K functions	5,844.67	5,876.67	3,933
CSmith 10K functions	20,190.90	3,649.04	19,161
LDRgen 530K functions	1,950.54	1,216.82	2,008
LDRGen 10K functions	4,753.50	322.65	4,668
DeepSmith 1K funcs	13.00	2.98	12
Actual Benchmarks	6,737.35	41,262.08	584

Fig. 7. Instructions per benchmarks in the collections that we distribute. SD is Standard Deviation.

The Angha collection contains 529.498 programs. As we shall explain in Section 4.1, we have already mined over a million compilable benchmarks using the infrastructure described in Section 3.1. However, we found that leaving an 1M-files tarball in our server was causing very long streaming times. To mitigate this issue, we provide half-a-million programs in a single compressed file. To reach this final number, 530K files, we used the following procedure:

(1) Let R be a list with the 100 largest git repositories with a majority of files in the C programming language, in descending order, and let C be the collection of benchmarks.
(2) While C has less than half-a-million files, we:
   (a) Remove r, the current largest repository from R;
   (b) Add to C every function from r, using the methodology in Section 3.1.

This algorithm stopped with 79 repositories, which include, for instance, the Linux kernel, git itself and FFmpeg.

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 2.1. 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. [26], which are themselves based on the work of Namolaru et al. [44]. We use Filho *et al.*'s features, instead of Namolaru's, because the latter is defined over C syntax, and, like Filho *et al.*, we run our feature extractor onto LLVM bytecodes.
Default: A default seven-features vector for fast searches: *number of instructions*, *number of stores*, *number of loads*, *number of basic blocks*, *number of CFG edges*, *number of SSA variables* and *number of variable uses*

Our similarity search does not relate programs based on semantic equivalence, à la Alon et al. [2]. Rather, close programs, in our context, are codes that tend to behave similarly when exposed to the same set of compiler optimizations. Example 3.4 illustrates this notion of similarity.

Example 3.4. Figure 8 shows results of a similarity search using numerical features vectors. The test program is the C implementation of insertion sort available in the Rosetta Code website[3]. For the sake of space, we show only the main loop of the query and result programs.

Query
```
1 rosettacode.org/wiki/Sorting_algorithms/Insertion_sort#C
2 for(size_t i = 1; i < n; ++i) {
3   int tmp = a[i];
4   size_t j = i;
5   while(j > 0 && tmp < a[j - 1]) {
6     a[j] = a[j - 1]; --j;
7   }
8   a[j] = tmp;
9 }
```

Result
```
11 ffmpeg.org/doxygen/0.6/wmavoice_8c-source.html
12 for (n = 0; n <= aidx; pulse_start++) {
13   for (idx = pulse_start; idx < 0; idx += fcb->pitch_lag) {
14     if (use_mask[idx >> 4] & (0x8000 >> (idx & 15))) {
15       use_mask[idx >> 4] &= ~(0x8000 >> (idx & 15));
16       n++;
17       start_off = idx;
18     }
19   }
20 }
```

Fig. 8. Main loops of programs related by similarity search using numerical feature vectors.

[3] http://rosettacode.org/

4 EVALUATION

This section investigates the following research questions[4]:

- **RQ1**: what is the current mining rate of the ANGHA infrastructure, described in Section 3.1?
- **RQ2**: what are the static properties typically found in programs available in open-source repositories?
- **RQ3**: can ANGHA better approximate the properties of human-written benchmarks than code produced by other program synthesizers?
- **RQ4**: can ANGHA better predict the impact of compiler optimizations onto real-world programs?
- **RQ5**: how effective is ANGHA, when used as the training set in predictive compilation, when compared to other synthetic benchmark collections?
- **RQ6**: Can we use ANGHA to train a predictive binary reduction tool that is competitive with a state-of-the-art binary reducer?

Ground Truth. RQ3 and RQ4 presuppose 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 endeavour 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.

4.1 RQ1: Mining Throughput

The *throughput* of the infrastructure described in Section 3.1 is the rate in which it produces valid benchmarks. A benchmark is considered valid when we can use both clang and gcc to convert it to an object file. In this section, we evaluate the throughput of our system.

Methodology: We set up our framework to collect and reconstruct code from the most popular C repositories in GitHub, 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 [41]. 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. The exact number of compilable programs generated was 1,044,023, and the entire process took approximately 145 hours to terminate. This gives us an average rate of one program generated per 0.5 second. In total, 1,882,687 candidate functions were extracted. Thus, the success rate for the reconstruction process was approximately 55.5%. While 5 seconds initially seemed too little time for the unification algorithm to run, we found that only 3,666 reconstructions failed at this step. The most common culprits for failures were errors during PsycheC's parsing process, due to unpreprocessed macros that were not syntactically valid in C. We found the distribution of number of programs per repository to be somewhat dominated by the largest projects. For instance, the Linux and FreeBSD kernels were the source of approximately 360k and 170k programs, respectively. Thus, these two alone account for more than half of the resulting collection.

[4]Further experiments are available as Supplementary Material.

4.2 RQ2: Static Properties

As previously mentioned in Section 1, one of the main benefits of having a large collection of compilable C functions is the possibility to analyze them statically. In this section, we abuse this possibility to inspect properties from real-world code. Some of these properties are folk knowledge. For instance, Boissinot et al. [12] say about SPEC CPU2006: "*About 95% percent of all variables have less than five uses. Over 70% of all variables have only one use. However, there are also cases in which variables have more than 600 uses.*" Angha lets us evaluate such statements in a much larger code base.

Methodology: To analyze programs, we convert them to the LLVM intermediate representation, and use LLVM's passes to obtain data of interest. We use the `mem2reg` pass to move into virtual registers the scalar values in the program. Using said methodology, this section analyzes four ratios involving static features of programs:

- Number of stores vs number of loads;
- Number of basic blocks vs number of instructions;
- Number of definitions vs number of uses;
- Number of basic blocks vs number of CFG edges.

Discussion: Figure 9 shows relations between the static features previously mentioned for the 530K compilable C functions available in the Angha repository. Each feature contains regression lines for the Angha programs, and for the ground-truth collection. We show the latter to contrast Angha with other synthetic collections. Figure 5 has already provided some evidence that these collections tend to produce programs too distant from typical C benchmarks. Each scatterplot in Figure 9 shows the slope of the regression line. This information lets us conclude, for instance, that, on average, each program variable (in the SSA representation) is used $1/0.594 = 1.68$ times, and each basic block tends to contain $1/0.146 = 6.85$ instructions.

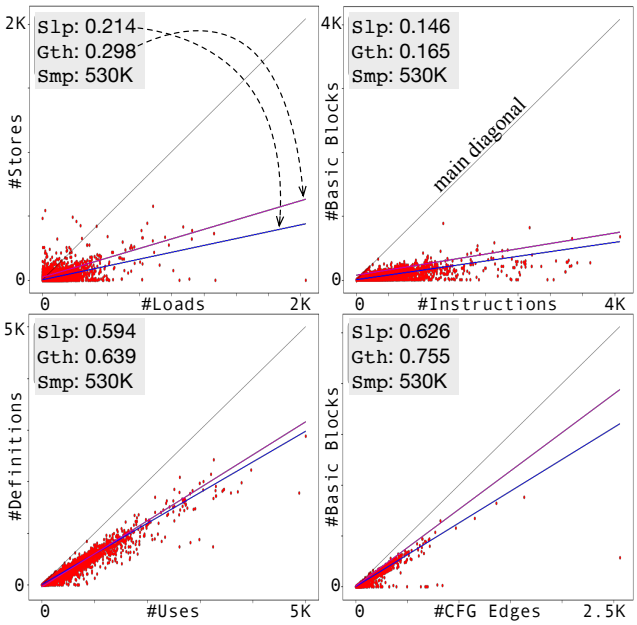

Fig. 9. Ratios found for the 530K compilable C functions available in the Angha repository. `Slp` is the regression slope for Angha, and `Gth` is the regression slope for the ground-truth collection.

Figure 9 reports results for individual functions; however, Angha also contains a collection of 15K compilable C files. Although we do not report the analysis of this collection in this paper, results are similar to those seen in Figure 9. Additionally, the static properties of the 10K largest programs in the Angha distribution also follow these same trends.

4.3 RQ3: Code Similarity

This paper defends the thesis that Angha 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.

Methodology: There exists a rich assortment of functions to measure the distance between programs [45]. 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. [26]. 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 3.2 already simplifies the computation of these two metrics. Furthermore, these two functions met the following properties, assuming that p_1 and p_2 are programs: (i) $d(p_1, p_2) \geq 0$, (ii) $d(p_1, p_2) = 0$ if, and only if, $p_1 = p_2$, (iii) $d(p_1, p_2) = d(p_2, p_1)$, and (iv) $d(p_1, p_2) \leq d(p_1, p_3) + d(p_3, p_2)$, for any $p_3 \notin \{p_1, p_2\}$.

Discussion: Figure 10 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.

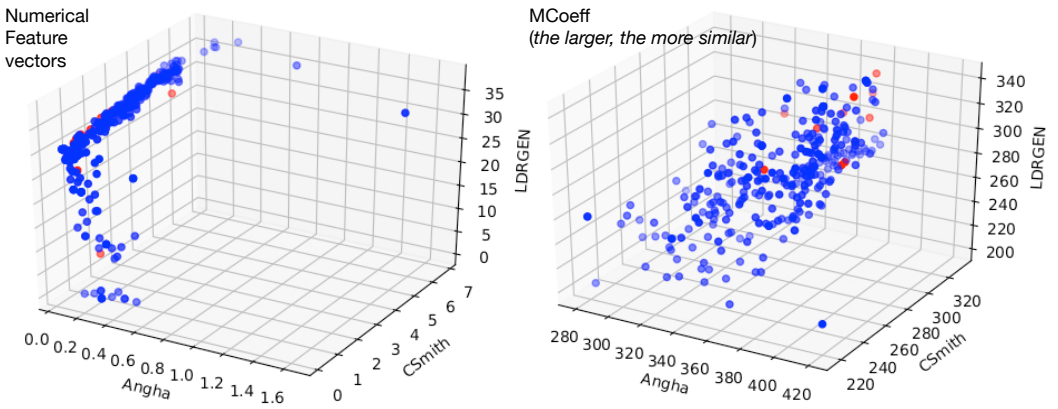

Fig. 10. Distance between the ground-truth and different synthetic benchmark collections. Each collection is formed by the 10K largest programs out of a pool of 530K candidates. Red dots are programs from the SPEC CPU2006 suite.

The average Euclidean distance from the ground-truth collection to Angha 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. 3.2) collection. This difference is smaller once we consider MCoeff, but it is still noticeable. Angha 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 demonstrate another fact: Angha is more diverse than the other synthetic collections that we use. The data in Figure 11 supports this statement. For each point $(N_b, K)_{(c,d)}$ in Figure 11, we assume that c is either Angha, CSmith, or LDRGen, and d is a distance function, e.g., numerical features, or MCoeff. In this case, N_b

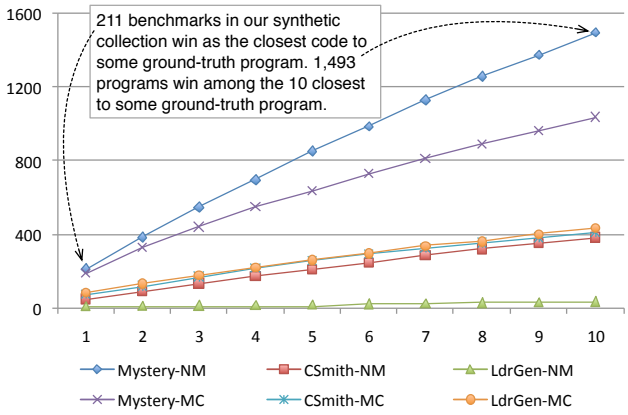

Fig. 11. Number of programs that win for K-nearest of some program in the ground-truth. The X-axis is K.

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.

4.4 RQ4: Predicting Compiler Behavior

The goal of this section is to support the thesis that Angha 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 onto different benchmark collections. To this end, we shall use the *Mean Square Prediction Error* (MSPE) as a measure of accuracy. MSPE is defined as $(predictedvalue - observedvalue)^2$. To carry out predictions, we fit a linear model M onto a synthetic benchmark, relating the size of programs when compiled with clang -O0 and clang -O1 (or -O3). We then use M to predict program size of optimizations on the ground-truth collection.

Discussion: Figure 12 relates program sizes, considering different benchmark collections, and different optimization levels. 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. Each figure also shows the slope of the regression line (`Slp`), and a measure of accuracy (`Err`). We let `Err` denote the ratio between the MSPE of a given collection (Angha, CSmith, LDRGen, DeepSmith) and the MSPE of the ground-truth suite. We compute it 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 onto the ground-truth collection (this is the standard definition of MSPE).
(4) Let m be the MSPE that we obtain when using ℓ also onto the ground-truth collection.
(5) We let `Err` $= m/m_g$.

Therefore, the lower the value of `Err`, the better is the predictor used to compute it. We report the error for two different collections in the Angha distribution: the first is formed by 530K single functions; the second is formed by 15K whole C files. As Figure 12 shows, Angha's error, regardless

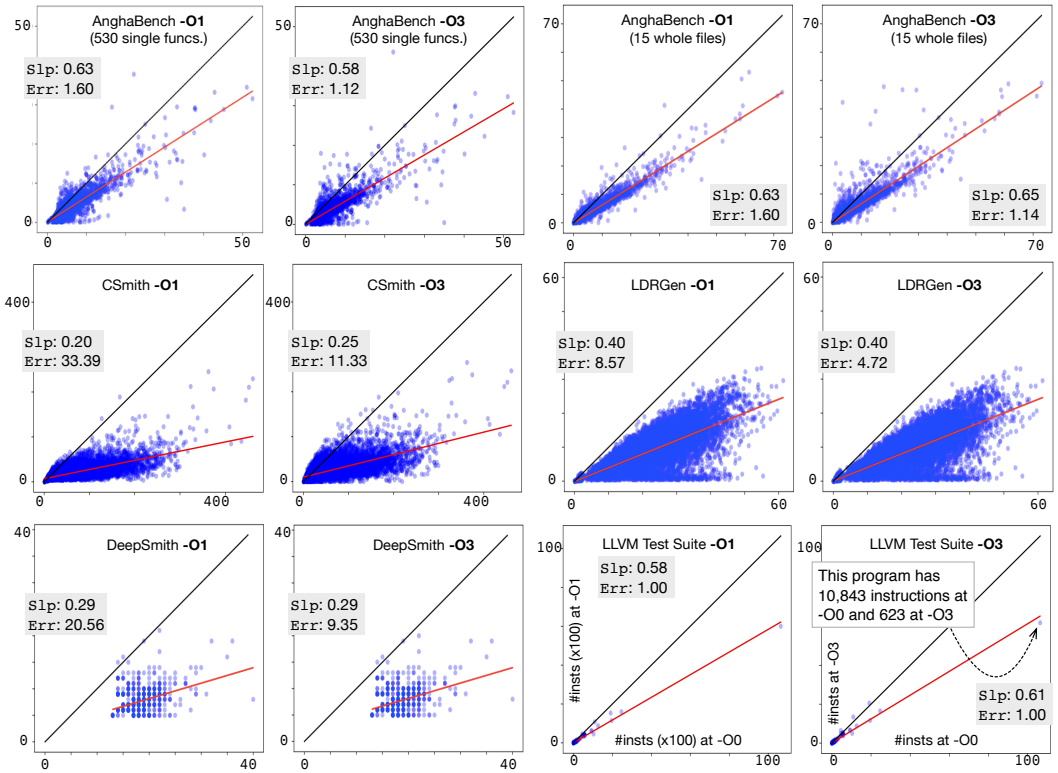

Fig. 12. The effect of compiler optimizations on the size of programs. Instructions are measured in hundreds.

of the collection used, is one order of magnitude less than the error produced by the other collections. In general, programs from the CSmith and from the LDRGen collections are easy to optimize. These programs are made to be executed without any undefined behavior. To avoid undefinedness, the inputs are hardcoded in the programs. This excess of constants leads to code that is easy to simplify via a combination of constant propagation and dead-code elimination. The programs generated by DeepSmith are also easy to optimize, although they do not contain hard-coded inputs (they are not meant to be executed). In this case, optimization opportunities come from an excess of dead-code.

4.5 RQ5: Predictive Compilation

This section provides evidence that Angha 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. [26]. We chose this compiler simply because its authors were kind enough to help us to set it up. YaCoS uses a heuristic based on Kennedy and Eberhart [35]'s particle swarm optimization (PSO) to find good optimization sequences.

Methodology: We use YaCoS to find sequences of 60 optimizations. Any permutation of the known universe of optimizations is acceptable. Said universe consists of 83 LLVM optimizations. Searching the feature space, in this setting, is the problem of associating with the feature vector of a program P the best list of optimization 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.

Once training is complete, YaCoS uses KNN to find the programs in the training set that are the closest to a given input program. We have used the two measures of distance seen in Section 4.3: Euclidean distance applied onto numerical feature vectors, and MCoeff. Thus, given an unknown program P_u, YaCoS finds the K programs that most resemble P_u; the list of optimizations of these programs is grouped in a set S_k. From S_k, YaCoS builds several list of optimizations chosen according to four different strategies:

Elite: we let $S_e \subseteq S_k$ be a subset of $S_k, K = 100$ formed only by the lists that yield smaller code than clang -Oz. The list of optimization of P_u is the average of the lists of programs in S_e.

J*X*, $X \in \{1, 10, 100\}$**:** we apply onto P_u the average of **J**ust the X best lists of optimizations in S_{100}.

We have trained YaCoS using the three collections mentioned in Section 3.2 with 10K files: Angha, CSmith and LDRGen. 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. For validation, we use the ground-truth collection mentioned in Section 4.3.

Discussion: Figures 13 and 14 summarize the results of this experiment for the Euclidean and MCoeff distances, respectively. A *winning strategy* is the pair in $\{\text{Elite}, \text{J1}, \text{J10}, \text{J100}\} \times \{\text{Angha}, \text{CSmith}, \text{LDRGen}\}$ that yields the smallest bytecodes when used to optimize the validation set. We omit results involving benchmarks produced by DeepSmith, because, due to their simplicity, their feature vectors contain mostly zeros. Boldface fonts mark winners, considering different metrics: minimum, maximum, mean and median code reduction. Angha wins in most cases. When using the Euclidean Distance with the Elite choice, Angha 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 collection used as validation contains the 13 integer programs from SPEC CPU2006.

	Mystery				CSmith				LDRGen			
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
	Elite	J1	J10	J100	Elite	J1	J10	J100	Elite	J1	J10	J100

Fig. 13. Percentage of code reduction 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.

The gray cells in Figures 13 and 14 contain only results for programs in the validation set for which we could find a list of optimization better than clang -Oz. **#P** is the number of such programs. If we consider average values for only these programs, then the difference between Angha and the other collections is less apparent, as we are counting only positive results. However, Angha is able to find non-trivial lists of optimization for substantially more programs than the other synthetic collections. For instance, using the Elite strategy with Euclidean distance (Fig. 13), YaCoS trained

with Angha 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.

	Mystery				CSmith				LDRGen			
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
	Elite	J1	J10	J100	Elite	J1	J10	J100	Elite	J1	J10	J100

Fig. 14. Percentage of code reduction achieved from the ground truth using YaCoS parameterized with the MCoeff distance.

4.6 RQ6: Code Size Reduction

This section provides some perspective on the code size reduction achieved by a compiler trained with Angha. To this end, we analyze the effects of this compiler onto MiBench [30]. This benchmark suite is of particular interest to this paper for two reasons: (i) this is a collection of programs meant to be deployed onto embedded devices; hence, size is a matter of concern for these benchmarks; (ii) MiBench has been used by Rocha et al. [51] 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 Angha. Program similarity is measured with the Euclidean distance. We use the subset of MiBench available in the LLVM test suite.

Discussion: Figure 15 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 15 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.

The amount of code size reduction obtained by YaCoS is considerably higher than the reduction achieved by FMSA; however, the bad performance of FMSA on MiBench had already been noticed by Rocha et al. [51]. FMSA does better on larger code sizes. Rocha *et al.* report averages of 6%. Our best setup gives us an average reduction of 11.1% (see Fig. 13). Nevertheless, 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 its more aggressive counterpart, 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.

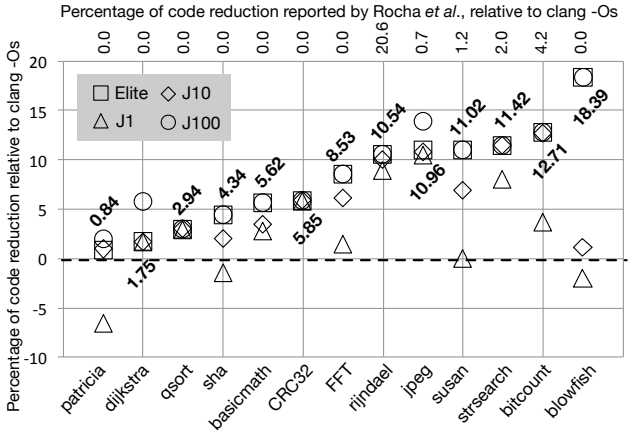

Fig. 15. Percentage of code size reduction achieved on MiBench. Numbers on top are percentage of reduction on object files reported by Rocha et al. [51]. Numbers at the bottom are percentage of reduction achieved by YaCoS/Elite.

5 RELATED WORK

This paper deals with synthesis of benchmarks and code size reduction. Because the former has been extensively discussed in Sections 1 and 2.3, we now focus on the latter. Yet, we start our discussion mentioning some work on the synthesis of benchmarks that we have not presented before.

Synthesis of Benchmarks. The creation of synthetic benchmarks has become a frequent focus of research. Random program generators, such as CSmith [62], LDRGen [9] and Orange3 [42, 43] have been successfully used to produce C programs for stress-testing compilers, often finding correctness bugs in industrial implementations. While originally conceived to find bugs, these tools have also been used to improve the quality of the optimized code emitted by mainstream C compilers [10, 32]. Nevertheless, such tools, given their goals, are not designed to produce realistic code.

Recent effort to create human-like programs has leveraged Deep Learning techniques to generate code similar to real-world examples. CLGen [20] uses this approach to generate OpenCL kernels, while DeepSmith [19] 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. We could not use it to train YaCoS, in Section 4.5, because the feature vectors of its programs contained mostly zeros.

Code Size Reduction. Several compiler transformations have code reduction as either their main goal or a desirable consequence [13, 16, 24, 36, 57]. One major category of such optimizations employ *Code Factoring*, which involves the identification of redundant code within the program. Code motion techniques search for identical instructions and move or merge them to avoid redundancy [13, 22, 40, 40]. *Function Merging* is the other main category in the field. These optimizations find functions which are semantically similar or equivalent, and generate new functions that substitute them. [34, 39, 56]. 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 [23, 51].

The contributions of this paper are orthogonal to these code size reduction techniques, because we do not propose new optimizations. Rather, our synthetic benchmarks can help a compiler identify when the use of each of these optimizations might be profitable. To support this observation, in Section 4.6 we showed how to augment a C compiler with knowledge extracted from our benchmarks; hence, allowing it to outperform the state-of-the-art code reduction technique of Rocha et al. [51].

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 [17, 25, 37]. 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 4.6, 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.

6 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. We ensure compilation by inferring missing dependences via Hindley-Milner's algorithm. This paper has presented several applications of these benchmarks, mostly concerning the training of predictive compilers. In this regard, we found it surprising that a mainstream compiler like clang still leaves much room for code size reduction, both at the -Os and at the -Oz levels. For instance, in benchmarks like `bitcount` and `blowfish`, both available in MiBench, it is possible to find sequences of optimizations that are about 15% better than clang -Os, and approximately 8% better than clang -Oz. In addition to these applications, we are aware of at least three other uses of Angha, outside our group. This collection of benchmarks has been used to measure the effectiveness of: (i) routing algorithms that convert C programs to FPGAs; (ii) auto-tuned register assignment heuristics; and (iii) C parsers synthesized from examples. Thus far, we have used Angha only statically, that is, without executing its programs. Therefore, a natural sequence of this work is the design of techniques to ensure the absence of undefined behavior when Angha's programs run.

REFERENCES

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

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[3] 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

[4] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *TACO* 13, 2 (2016), 21:1–21:25. https://doi.org/10.1145/2928270

[5] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. 1998. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer* 31, 10 (1998), 77–85. https://doi.org/10.1109/2.722293

[6] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.* 5, 3 (1991), 63–73. https://doi.org/10.1177/109434209100500306

[7] Antoine Balestrat. 2016. CCG: A Random C code generator. https://github.com/Mrktn/ccg

[8] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *OSDI*. USENIX Association, Berkeley, CA, USA, 177–192.

[9] Gergö Barany. 2017. Liveness-Driven Random Program Generation. In *LOPSTR*. Springer, Heidelberg, Germany, 112–127. https://doi.org/10.1007/978-3-319-94460-9_7

[10] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 82–92. https://doi.org/10.1145/3178372.3179521

[11] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A Black-box Approach to Energy-aware Scheduling on Integrated CPU-GPU Systems. In *CGO*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/2854038.2854052

[12] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dine hin, and Fabri e Rastello. 2008. Fast Liveness Checking for Ssa-form Programs. In *CGO*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1356058.1356064

[13] Preston Briggs and Keith D. Cooper. 1994. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 159–170. https://doi.org/10.1145/178243.178257

[14] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. 2017. Learning to superoptimize programs. In *ICLR*. OpenReview, Toulon, France, Article 1, 14 pages.

[15] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[16] John Cocke. 1970. Global Common Subexpression Elimination. *SIGPLAN Not.* 5, 7 (July 1970), 20–24. https://doi.org/10.1145/390013.808480

[17] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/314403.314414

[18] T. Cover and P. Hart. 2006. Nearest Neighbor Pattern Classification. *Trans. Inf. Theor.* 13, 1 (2006), 21–27. https://doi.org/10.1109/TIT.1967.1053964

[19] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *ISSTA*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[20] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, Piscataway, NJ, USA, 86–99.

[21] 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.

[22] A. Dreweke, M. Worlein, I. Fischer, D. Schell, Th. Meinl, and M. Philippsen. 2007. Graph-Based Procedural Abstraction. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 259–270. https://doi.org/10.1109/CGO.2007.14

[23] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting Function Similarity for Code Size Reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/2597809.2597811

[24] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. 1997. Code Compression. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 358–365. https://doi.org/10.1145/258915.258947

[25] D. Fatiregun, M. Harman, and R. M. Hierons. 2004. Evolving transformation sequences using genetic algorithms. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. 65–74. https://doi.org/10.1109/SCAM.2004.11

[26] 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

[27] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International journal of parallel programming* 39, 3 (2011), 296–327.

[28] Grigori Fursin and Olivier Temam. 2010. Collective Optimization: A Practical Collaborative Approach. *Trans. Archit. Code Optim.* 7, 4 (2010), 20:1–20:29. https://doi.org/10.1145/1880043.1880047

[29] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub Data on Demand. In *MSR*. ACM, New York, NY, USA, 384–387. https://doi.org/10.1145/2597073.2597126

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC*. IEEE, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[31] Tianyi David Han and Tarek S. Abdelrahman. 2017. Use of Synthetic Benchmarks for Machine-Learning-Based Performance Auto-Tuning. In *IPDPS Workshops*. IEEE Press, Piscataway, NJ, USA, 1350–1361.

[32] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology* 9 (2016), 21–29.

[33] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[34] LLVM Compiler Infrastructure. 2019. MergeFunctions pass, how it works. Retrieved November 10, 2019 from http://llvm.org/docs/MergeFunctions.html#mergefunctions-pass-how-it-works

[35] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of the International Conference on Neural Networks*, Vol. 4. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[36] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. *SIGPLAN Not.* 29, 6 (June 1994), 147–158. https://doi.org/10.1145/773473.178256

[37] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. 2003. Finding Effective Optimization Phase Sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES '03)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/780732.780735

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[39] Martin Liška. 2014. *Optimizing large applications*. Master's thesis. arXiv:cs.PL/1403.6997

[40] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. 2004. Code factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*. 79–84.

[41] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2018. Inference of Static Semantics for Incomplete C Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 29 (Dec. 2018), 28 pages. https://doi.org/10.1145/3158117

[42] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Trans. System LSI Design Methodology* 7 (2014), 91–100.

[43] Kazuhiro Nakamura and Nagisa Ishiura. 2015. 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)*. 226–227.

[44] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

[45] Rashid Naseem, Onaiza Maqbool, and Siraj Muhammad. 2011. Improved Similarity Measures for Software Clustering. In *CSMR*. IEEE Computer Society, Washington, DC, USA, 45–54. https://doi.org/10.1109/CSMR.2011.9

[46] Marcelo Novaes, Vinícius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quintão Pereira. 2019. Compiler-assisted Adaptive Program Scheduling in Big.LITTLE Systems: Poster. In *PPoPP*. ACM, New York, NY, USA, 429–430.

[47] Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. 2018. Static Prediction of Silent Stores. *ACM Trans. Archit. Code Optim.* 15, 4, Article 44 (2018), 26 pages. https://doi.org/10.1145/3280848

[48] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *ASPLOS*. ACM, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

[49] Gabriel Poesia, Breno Campos Ferreira Guimarães, Fabricio Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static placement of computation on heterogeneous devices. *PACMPL* 1, OOPSLA (2017), 50:1–50:28.

[50] Louis-Noël Pouchet and Tomofumi Yuki. 2018. PolyBench/C 4.2.1: The polyhedral C benchmark suite. http://polybench.sf.net

[51] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *CGO*. IEEE Press, Piscataway, NJ, USA, 149–163.

[52] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (2016), 114–122. https://doi.org/10.1145/2863701

[53] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *CGO*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[54] Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. 2018. CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors. *Trans. Multi-Scale Computing Systems* 4, 2 (2018), 163–176.

[55] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 77–90. https://doi.org/10.1145/781131.

781141

[56] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit*. http://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=tallam.pdf

[57] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. 1982. Using Peephole Optimization on Intermediate Code. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 21–36. https://doi.org/10.1145/357153.357155

[58] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *PPoPP*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/1504176.1504189

[59] Zheng Wang and Michael F.P. O'Boyle. 2010. Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach. In *PACT*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/1854273.1854313

[60] 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

[61] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *HiPC*. IEEE, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/HiPC.2014.7116910

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[63] Peng Zhao and José Nelson Amaral. 2003. To Inline or Not to Inline? Enhanced Inlining Decisions. In *LCPC*. Springer, Heidelberg, Germany, 405–419.