Synthesis of Benchmarks for the C Programming Language by Mining Software Repositories

Breno Campos Ferreira Guimarães (UFMG)
José Wesley Magalhães (UFMG)
Anderson Faustino da Silva (UEM)
Fernando Magno Quintão Pereira (UFMG)
MOTIVATION
Compilers apply many **heuristics** to parameterize their behaviour

- How many loop iterations to unroll?
- How many function calls to inline?
- Which variables to spill during register allocation?
- Etcetera, etcetera...
Compilers apply many heuristics to parameterize their behaviour:

- How many loop iterations to unroll?
- How many function calls to inline?
- Which variables to spill during register allocation?
- Etcetera, etcetera...

These are all hard problems!
Compiler Autotuning

- Let computers pick the best parameters:
• Let computers pick the best parameters:

Bunch of Programs (a.k.a "Training Set")
Let computers pick the best parameters:

- Compiler Autotuning

Bunch of Programs (a.k.a "Training Set")

Compiler
• Let computers pick the best parameters:

Bunch of Programs (a.k.a "Training Set") → Compiler → Evaluate Metrics (performance, code size, whatever)
• Let computers pick the best parameters:

Bunch of Programs (a.k.a "Training Set")

Compiler

Evaluate Metrics (performance, code size, whatever)

Machine Learning Model
Compiler Autotuning

- Let computers pick the best parameters:

  New, unknown program (a.k.a "Test Set")
Let computers pick the best parameters:

- New, unknown program (a.k.a "Test Set")
- Machine Learning Model
● Let computers pick the best parameters:

New, unknown program (a.k.a "Test Set") → Machine Learning Model → (Hopefully) Better Optimized Program → Compiler
The Need for Data

(Hopefully)
Better
Optimized
Program
The Need for Data

(Hopefully)
Better
Optimized
Programs

Which programs??

Bunch of Programs
(a.k.a "Training Set")
The Need for Data

- State-of-the-art autotuning approaches have used an average of 17 (seventeen!) programs!¹

The Need for Data

- State-of-the-art autotuning approaches have used an average of 17 (seventeen!) programs!
- Our goal: synthesize benchmarks that approximate real-world code.

PREVIOUS APPROACHES
Synthesizers (not the music kind)

- Some previous works focus on creating synthetic programs:
Synthesizers (not the music kind)

- Some previous works focus on creating synthetic programs:
  - **CSmith**\(^1\) - generates **random** syntactically valid C programs

---

Some previous works focus on creating synthetic programs:

- **CSmith**\(^\text{¹}\) generates random syntactically valid C programs.


However, the random programs are distant from usual code: unused variables, trivially dead code, etc.
Synthesizers (not the music kind)

- Some previous works focus on creating synthetic programs:
  - Csmith\(^1\)
  - ldrgen\(^2\) - generates random programs, but focusing on liveness

---

\(^1\) - Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. *Finding and understanding bugs in C compilers*. SIGPLAN Not. 46, 6 (June 2011), 283-294. DOI: [https://doi.org/10.1145/1993316.1993532](https://doi.org/10.1145/1993316.1993532)

Some previous works focus on creating synthetic programs:

- Csmith¹
- ldringen² - generates random programs, but focusing on liveness

This mitigates the dead code issue, but programs are still random, and thus not very realistic.

Some previous works focus on creating synthetic programs:

- Csmith\textsuperscript{1}
- ldrgen\textsuperscript{2}
- CLGen/DeepSmith\textsuperscript{3,4} - Use Deep Learning to replicate real-world code.

\textsuperscript{1} - Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. \textit{Finding and understanding bugs in C compilers}. SIGPLAN Not. 46, 6 (June 2011), 283-294. DOI: \url{https://doi.org/10.1145/1993316.1993532}
Some previous works focus on creating synthetic programs:

- Csmith
- ldrgen
- CLGen/DeepSmith

- Use Deep Learning to replicate real-world code.

Model training/program generation are very time-consuming, and benchmarks are still trivial.


# The Benchmark Synthesis Map

<table>
<thead>
<tr>
<th>Method</th>
<th>Goal</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Csmith</td>
<td>Stress-testing compilers</td>
<td>Random programs, not realistic</td>
</tr>
<tr>
<td>Isgren</td>
<td>Stress-testing compilers</td>
<td>Slightly more realistic programs, but still not representative</td>
</tr>
<tr>
<td>CLGen</td>
<td>Feed ML models to optimize OpenCL kernels</td>
<td>Restricted to OpenCL, and involves time-consuming dataset creation/curating and long training/generation cycles</td>
</tr>
<tr>
<td>DeepSmith</td>
<td>Fuzz-test compilers</td>
<td>Involves time-consuming processing, and generates very trivial programs</td>
</tr>
</tbody>
</table>
So why not simply take big open-source code and use them as benchmarks?

- Very difficult, for several reasons:
  - Large codebases
  - Complex build systems
  - Dependencies (libraries, other modules, etc)
The project

● So why not simply take big open-source code and use them as benchmarks?
  ○ Very difficult, for several reasons:
    ■ Large codebases
    ■ Complex build systems
    ■ Dependencies (libraries, other modules, etc)
  ● Thus, downloading public code and compiling it = impractical!
So why not simply take big open-source code and use them as benchmarks?

- Very difficult, for several reasons:
  - Large codebases
  - Complex build systems
  - Dependencies (libraries, other modules, etc)

Thus, downloading public code and compiling it = impractical!

Not anymore!
The project

● We developed a **completely automated** methodology to:
  ○ Mine public large C codebases
  ○ Extract code at a function-by-function granularity
  ○ Generate compilable versions of each function, which can be used as standalone programs for modeling
The project

- Used it to build **AnghaBench**:  
  - ~530,000 *compilable* C programs, of varying sizes
Methodology

Our approach has three main components:
Our approach has three main components:

- **Code Miner**
- **C Code Corpus**
- **Function Extractor**

**Compilable Functions**

**Code Reconstructor**

**(incomplete) C Functions**
To mine repositories, we built a simple crawler that leverages GitHub’s API to download code.

- Downloaded ~80 of the largest C codebases, including:
  - The Linux Kernel
  - FFmpeg
  - Git
  - curl
  - etc.
Our approach has three main components:

- Code Miner
- C Code Corpus
- Function Extractor

- Compilable Functions
- Code Reconstructor
- (incomplete) C Functions
To extract functions, we developed a plugin for Clang, LLVM’s C language frontend.
To extract functions, we developed a plugin for Clang, LLVM’s C language frontend.
To extract functions, we developed a plugin for Clang, LLVM’s C language frontend.

Function Extractor

- Translation Unit (C Source File) → Clang
- Clang → (probably incomplete) Abstract Syntax Tree
- Find function definition nodes and outline them
To extract functions, we developed a plugin for Clang, LLVM's C language frontend.
Function Extractor

- Extracted almost 700,000 C functions.
Our approach has three main components:

- Code Miner
- C Code Corpus
- Function Extractor
- Compilable Functions
- Code Reconstructor
- (incomplete) C Functions
To reconstruct missing types, we use psyche-c¹, a Hindler/Milner type inference engine for C.
Code Reconstructor

(this is real code, taken from the toxcore¹ repo)

```c
int bs_list_find(const BS_LIST *list, const uint8_t *data) {
    int r = find(list, data);

    //return only -1 and positive values
    if (r < 0) {
        return -1;
    }

    return list->ids[r];
}
```

Code Reconstructor

Missing types!

```c
int bs_list_find(const BS_LIST *list, const uint8_t *data)
{
    int r = find(list, data);
    //return only -1 and positive values
    if (r < 0) {
        return -1;
    }
    return list->ids[r];
}
```
int bs_list_find(const BS_LIST *list, const uint8_t *data) {
  int r = find(list, data);

  //return only -1 and positive values
  if (r < 0) {
    return -1;
  }

  return list->ids[r];
}
```c
typedef int uint8_t;

struct TYPE_4__ { int* ids; };

typedef struct TYPE_4__ BS_LIST;

int find(BS_LIST const*, int const*);

int bs_list_find(const BS_LIST *list, const uint8_t *data) {
    int r = find(list, data);
    //return only -1 and positive values
    if (r < 0) {
        return -1;
    }

    return list->ids[r];
}
```
Code Reconstructor

typedef int uint8_t;
struct TYPE_A { int* ids; }
typedef int find (TYPE_A *a, int bs_list [4], int8_t *data) {
    int r = -1;
    //return only -1 and positive values
    if (r < 0) {
        return -1;
    }
    return list->ids[r];
}
Code Reconstructor

- Out of the ~700,000 extracted functions, we generated ~530,000 compilable ones.
EVALUATION
Evaluation

• We analyzed the distribution of the 530k functions by measuring the number of LLVM instructions in each program
Evaluation

- We analyzed the distribution of the 530k functions by measuring the number of LLVM instructions in each program
Evaluation

- We analyzed the distribution of the 530k functions by measuring the number of LLVM instructions in each program.

Very skewed towards small programs
Evaluation

- We analyzed the distribution of the 530k functions by measuring the number of LLVM instructions in each program.

**Mean** number of instructions: 63.27
**Median** number of instructions: 36
Evaluation

- Due to the skewed nature of the data, we created a new population from the 10k largest programs.
Evaluation

- Due to the skewed nature of the data, we created a new population from the 10k largest programs.
Evaluation

- Due to the skewed nature of the data, we created a new population from the 10k largest programs.
Due to the skewed nature of the data, we created a new population from the 10k largest programs.

**Mean** number of instructions: 441.8

**Median** number of instructions: 360
Why AnghaBench is closer to the truth

- To evaluate the representativeness of AnghaBench, we compare the 10k largest functions to two other populations:
  - 10k Csmith generated kernels (*synthetic*)
  - 250 programs from the LLVM test-suite (*ground truth*)
- Compile all using -O0, -O1 and -O3
- Compare number of instructions in each (to evaluate number of instructions removed/added by optimization passes)
Why AnghaBench is closer to the truth

![Graphs showing the comparison between AnghaBench and other benchmarks. The graphs indicate that AnghaBench is closer to the truth.](image-url)
Why AnghaBench is closer to the truth

**RMSE** is 33.3x bigger than using LLVM data itself for regression!
Why AnghaBench is closer to the truth
Why AnghaBench is closer to the truth

RMSE is only 1.6x bigger. Much better fit!
Why AnghaBench is closer to the truth

[Scatter plots showing comparison between AnghaBench and CSmith for #instructions (10^4) with O3 and O0 optimization levels.]

AnghaBench - O3

#instructions (10^4) - O3

CSmith - O3

#instructions (10^4) - O0
Why AnghaBench is closer to the truth

AnghaBench predictions are 10.1x more accurate!
Takeaways

- Compilers use heuristics for autotuning
- This approach requires synthetic benchmarks
- Current program synthesizers are not good
- We have designed a new data synthesizer
- Code is downloaded from open-source repos
- Type inference ensures compilation
QUESTIONS

E-mail us:
Breno Campos Ferreira Guimarães [brenosfg@dcc.ufmg.br]
José Wesley Magalhães [jwdesmagalhaes@gmail.com]
Fernando Magno Quintão Pereira [fernando@dcc.ufmg.br]
Anderson Faustino da Silva [andersonfaustino@gmail.com]