Jotai: a Methodology for the Generation of Executable C Benchmarks

Cecília Conde Kind
UFMG
Brazil
cissakind@gmail.com

Fernando M. Quintão Pereira
UFMG
Brazil
fernando@dcc.ufmg.br

Abstract

This paper introduces a methodology to generate well-defined executable benchmarks in the C programming language. The generation process is fully automatic: C files are extracted from open-source repositories, and split into compilation units. A type reconstructor infers all the types and declarations required to ensure that functions compile. The generation of inputs is guided by constraints specified via a domain-specific language. This DSL refines the types of functions, for instance, creating relations between integer arguments and the length of buffers. Off-the-shelf tools such as AddressSanitizer and Kcc filter out programs with undefined behavior. To demonstrate applicability, this paper analyzes the dynamic behavior of different collections of benchmarks, some with up to 30 thousand samples, to support several observations: (i) the speedup of optimizations does not follow a normal distribution—a property assumed by statistical tests such as the T-test and the Z-test; (ii) there is strong correlation between number of instructions fetched and running time in x86 and in ARM processors; hence, the former—a non-varying quantity—can be used as a proxy for the latter—a varying quantity—in the autotuning of compilation tasks. The apparatus to generate benchmarks, plus a collection of 30K programs thus produced, is publicly available.

Keywords: Benchmark, Optimization, Compiler, Search

1 Introduction

Predictive compilation is a family of techniques whose goal is to let optimizing compilers treat programs differently. The predictive compiler is trained onto a large corpus of programs, determining, for each one of them, the sequence of analyses and optimizations that suits that code better. Once given an unknown program, the compiler uses the knowledge acquired during training to determine the best way to treat this new code. Predictive compilation methodologies have been known for many years [32, 33, 37, 51]. Nevertheless, the growing popularity of machine learning techniques have attracted new attention to this field, and much progress in the design and implementation of predictive compilers has been attained in the last five years [3, 8, 9, 13, 15, 18, 21, 22, 38, 42, 47, 53, 54].

The Need for Benchmarks. Training a predictive compiler requires benchmarks, as emphasized in Wang and O'Boyle [49]'s survey. Thus, there has been much recent research effort along the automatic synthesis of large collections of benchmarks. In 2016, Mou et al. [38] released Poj: 104 classes of programming problems, each with 500 solutions. Later, in 2017, Cummins et al. [14] produced CLgen, a tool that synthesizes OpenCL benchmarks. In 2021, da Silva et al. [18] released AnghaBench, a suite with more than one million compilable programs. A few months later, Puri et al. [43] released CodeNet. Like Poj, the CodeNet suite consists of solutions to programming problems; however, this collection is two orders of magnitude larger.

The Challenge: Sound Executable Code. Most of the research to create benchmarks orbit around C, or around similar languages, such as OpenCL. Programs written in these languages might present *undefined behaviors*: the execution of actions not defined by the standard semantics of the language [30]. Consequently, large collections of benchmarks [18, 38] reconstructed from open-source repositories are formed by programs that compile but do not run. To give the reader some perspective on the problem, notice that CompilerGym [15], a framework for the autotuning of compilation tasks, provided 1,158,701 benchmarks spread across 12 datasets in October 2022. However, before Jotai programs were incorporated to it, only 23 programs from CBench [25] would come with executable inputs.

Benchmarks generated to emulate human-made code have been released recently [2, 6, 7, 14, 47]. Two of these collections contained C code [2, 6], but only Berezov et al.'s CodeLaGen could be executed automatically. These are simple kernels — nests of loops that process arrays. Although simple, every program that we tried to run showed some form of undefined behavior, once compiled with Kcc [30]. We also compiled Armengol-Estapé et al.'s ExeBench with Kcc. In this case, compilation is not automatic: we had to manually fill up missing libraries. In spite of that, all the programs that we run contained undefined behavior. During our experience with these benchmark generators, we also had to deal with another limitation: they do not provide a way to steer the generation of program inputs. Inputs are hard-coded in the synthesizer: essentially, they consist of large buffers filled with random values. It is not possible, for instance, to establish relations between function arguments.

The Contributions of this Work. The goal of this paper is to propose a methodology to generate executable C benchmarks. This methodology exists around JotaiLang, a domain-specific language that we have designed to generate inputs for programs. The process to generate benchmarks relies on a number of techniques and tools:

- **Techniques:** JotaiLang lets developers impose constraints on the inputs that are randomly tried on each benchmark. Constraints are derived from the signature of the target function. This combination of types and constraints prunes the space of possible inputs; hence, focusing input generation on values that are more likely to result into well-defined executions.
- **Tools:** (i) a web crawler that retrieves C functions from GitHub; (ii) an off-the-shelf type inference engine for C that ensures that those functions compile [34, 35] (iii) a code generator that produces drivers to run each benchmark; (iv) a Valgrind plugin [45] that measures coverage of these inputs; and (v) Kcc [30] plus Asan [46] to detect undefined behaviors.

Throughout the process of generating benchmarks and interacting with people who use them, we have compiled a list of requirements that these programs must meet:

- **Compile-and-run:** each benchmark comes in a separate file as an independent compilation unit, with all the drivers necessary to run it.
- **Sound:** each benchmark abides by the semantics that Hathhorn et al. [30] have defined for the C programming language.
- **Deterministic:** the library that generates inputs is hard-coded in each benchmark, and uses a deterministic number generator.
- **Profilable:** benchmarks can contain multiple input sets. Some can be used for training and others for testing.
- **Visible:** Jotai benchmarks do not invoke library functions. Hence, every instruction is *visible* [1]. Thus, a sanitizer like Kcc can observe the execution of every instruction when looking for undefined behavior.
- **Observable:** every function that makes up Jotai returns a value. This output can be used as a way to find bugs in compilers and interpreters.
- **Clean:** Every memory allocated by that function's driver is deallocated before termination.

Summary of Results. We have made a collection of 36,223 executable programs mined from GPL repositories publicly available. Since October 2022, 18,761 of these functions are available as a CompilerGym dataset [15]. Nevertheless, Jotai benchmarks are extracted from open-source repositories, and there is no limit for how many programs can be created via the methodology that Section 3 introduces. Section 4 describes some uses of the Jotai collection. Section 4.1 shows that it is fair to expect the construction of a valid benchmark (no undefined behavior dynamically detected by Kcc) for each 34-35 functions that we find in C files from open-source repositories. Section 4.2 analyses the speedup of optimizations on different subsets of the Jotai collection and on Spec Cpu2017. Such speedups do not follow a normal distribution, which is assumed in several statistical tests. Section 4.3 observes a strong correlation between the running time of programs and the number of instructions fetched on Intel i7 and on ARM A15 processors. Correlation holds when programs are compiled with different optimization levels. Section 4.4 employs machine learning models to predict the 'speedup' effect of optimizations based on features extracted from source code. These features are represented as histograms, which capture the occurrences of opcodes of LLVM instructions in the programs [19]. By using these histograms as predictive indicators, accurate speedup prediction is demonstrated. Finally, Section 4.5 discusses structural properties of the programs. On average, functions tend to comprise four to six basic blocks, and about half the functions have their control-flow graphs fully covered by inputs that we generate.

2 The Anatomy of a Benchmark

Jotai benchmarks are produced out of programs mined from open-source repositories. The generation of benchmarks works by: (i) extracting C files via a web crawler; (ii) splitting functions in each C file into single files; (iii) inferring types for each benchmark file; (iv) generating a driver for each compilable benchmark file; (v) filtering out inputs that lead to undefined behavior. Figure 1 shows these different steps, and Example 2.1 illustrates the first steps of this process.

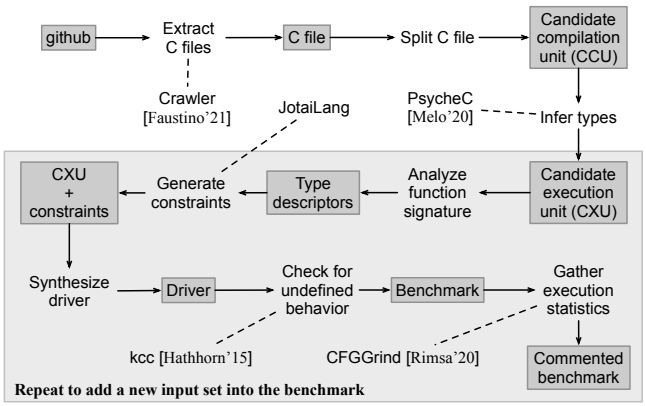

Figure 1. The benchmark generation process. This process is entirely automatic, and requires starting one single script.

Example 2.1. Figure 2 shows a function from `status.c`, a file taken from the `sqlite` repository. There are eight functions with a body within `status.c`. Jotai tries to produce a benchmark out of each one of them. The process starts with

code extraction: function `countLookasideSlots` is placed into a separate C file: the *candidate compilation unit* (CCU). Types and missing declarations are then inferred for this CCU via a tool called PsycheC [34, 35]. Figure 2 (b) shows the types inferred for function `countLookasideSlots`. If PsycheC terminates successfully, then the reconstructed program is guaranteed to compile. However, PsycheC might fail. In this case, the candidate compilation unit is discarded.

```
175 …
176 static u32 countLookasideSlots(LookasideSlot *p) {
177   u32 cnt = 0;
178   while( p ){
179     p = p->pNext;
180     cnt++;
181   }
182   return cnt;
183 }
184 …
```

(a)

```
(b) /* Forward declarations */
    typedef struct TYPE_3__  TYPE_1__;

    /* Type definitions */
    typedef  long u32 ;
    struct TYPE_3__ {
      struct TYPE_3__* pNext;
    };
    typedef  TYPE_1__ LookasideSlot;
```

Figure 2. (a) Code snippet taken from file `status.c` from the `sqlite` repository. (b) Types that PsycheC infers to ensure compilation of function `countLookasideSlots`.

Visible Instructions. Example 2.1 illustrates one of the principles of Jotai: benchmarks yield only *visible instructions*. Following Álvares et al. [1]'s terminology, given a program P with source code S, and a compiler C, the visible instructions of P are the instructions that C produces for statements that appear in S. Every other instruction required for the execution of P is an *invisible* instruction. Invisible instructions come from dynamically linked libraries and routines added by the compiler, such as initialization (pre-`main` code) and finalization (post-`main` code). To meet this visibility requirement, Jotai benchmarks are not allowed to invoke functions without bodies. This restriction serves two purposes. First, it eases the task of discovering undefined behavior during the execution of the program, as tools like Kcc or FramaC only have access to the visible part of a program. Second, it prevents the benchmark from invoking malicious code.

Drivers. A candidate compilation unit that compiles becomes a *candidate executable unit* (CXU). If Jotai succeeds in producing an input for a CXU that does not incur in undefined behavior, then this program becomes what we call a 1-input *driver*. These benchmarks can be augmented gradually. If Jotai succeeds in generating a new input for an n-input driver, then this program becomes an $(n+1)$-input driver. Input generation is steered by constraints, which, in turn, Jotai derives from the type signature of the target function. Constraint generation is the subject of Sections 3.1 and 3.2. For now, it suffices to know that each set of constraints that yields a well-defined execution contributes one input to the driver. Example 2.2 clarifies this terminology.

Example 2.2. Figure 3 shows a 2-input driver produced to run the function in Example 2.1. This driver contains a switch with two cases: each case feeds the function `countLookaside Slots` with different inputs. Everything in Figure 3 is synthesized automatically from the constraints in Section 3.2.1.

```
01  …
02  int main(int argc, char *argv[]) {
03    if (argc != 2) {
04      usage();
05      return 1;
06    }
07    int opt = atoi(argv[1]);
08    switch(opt) {
09      case 0: { // int-bounds
10        struct TYPE_3__ * aux_p[1];                          Generate inputs
11        struct TYPE_3__ * p = _allocate_p(1, aux_p);
12        long benchRet = countLookasideSlots(p);              Execute
13        printf("%ld\n", benchRet);                           Observe
14        _delete_p(aux_p, 1);                                 Clean up
15        break;
16      }
17      case 1: { // linked
18        struct TYPE_3__ * aux_p[10000];                      Generate
19        struct TYPE_3__ * p = _allocate_p(10000, aux_p);     inputs
20        long benchRet = countLookasideSlots(p);              Execute
21        printf("%ld\n", benchRet);                           Observe
22        _delete_p(aux_p, 10000);                             Clean up
23        break;
24      }
25      default:
26        usage();
27        return 2;
28    }
29    return 0;
30  }
31  …
```

Figure 3. The driver produced by Jotai to run function `countLookasideSlots`, seen in Figure 2.

Compile-and-Run. Example 2.2 illustrates some principles enumerated in Section 1. First, concerning compilation, assuming that this two-input driver is in a file called `driver.c`, we can compile it with simply "`clang/gcc driver-c`". Concerning execution, we can run the executable file with either the command "`./a.out 0`", or the command "`./a.out 1`". If invoked without arguments, then the driver outputs a usage guide with a brief explanation about each input set.

Profile. The driver seen in Example 2.2 features two sets of inputs. Often, Jotai benchmarks provide more than that. Multiple inputs let developers use the Jotai benchmarks in a profile-guided setting, where some inputs are used as training, and the others as testing.

Observe. The driver is also observable, meaning that it prints the result of the function. In this simple example, the output of interest is a simple scalar value; however, Jotai benchmarks can print the contents of aggregate types, such as instances of `struct` or `union` types. Notice that inspection is shallow: recursive types like linked lists or trees are not traversed. Jotai can also be configured to add timing routines to print the execution time of the target function; however,

this code is not portable across operating systems; hence, it is disabled by default.

Clean up. Every `case` of a driver ends with calls to a routine that frees allocated memory. This clean up is only necessary when benchmarking functions whose signature contains arguments of pointer type. Notice that JOTAI is able to generate recursive data structures, as in the second input set of Figure 3. Cleaning code will free every node that constitutes a recursive data structure. As a consequence, every JOTAI benchmark runs until normal termination (exit code 0) when compiled with an address sanitization; for instance, via `gcc -fsanitize=address`.

Deterministic behavior. JOTAI benchmarks are deterministic, meaning that the execution of an n-driver with a certain argument i, $1 \leq i \leq n$, will always lead to the execution of the same sequence of instructions. To ensure determinism, benchmarks are sequential programs: no multi-threading is allowed. Furthermore, the routines that generate inputs for the benchmark are hardcoded into the driver. These routines include code to produce scalars of every primitive type in the ANSI C language specification, and code to generate recursive data structures like lists and trees. Function `countLookasideSlots` in Example 2.1 contains one argument of a recursive type, as seen in Figure 2 (b). The second case in Figure 3 will generate a linked list with 10,000 nodes with this structure.

3 A Methodology to Generate Code

Figure 1 shows the steps to generate benchmarks. Part of this methodology (outside the gray box), has been already used in previous work [18], and we omit it from this presentation. The rest of it is the subject of this section.

3.1 Extraction of Type Descriptors

The constraints that JOTAI produces to guide the generation of inputs—to be explained in Section 3.2—relies on *type descriptors*. Type descriptors model the structure of the types of the arguments of the CCU function. Descriptors are specified by the grammar in Figure 4. Following C's static semantic, type descriptors determine a nominal type system, e.g., two types are the same if they share the same names.

typeDesc	::=	(typeStruct | typeFun)*
typeStruct	::=	`struct` <*name*> typeBindings
typeFun	::=	`function` <*name*> typeName typeBindings
typeBindings	::=	<*name*> typeName typeBindings | ε
typeName	::=	(`unsigned`)? typeScalar | `*` typeName | `struct` <*name*>
typeScalar	::=	`char` | `int` | `short` | `long` | `float` | `double`

Figure 4. The grammar to specify type descriptors.

Example 3.1. Figure 5 (a) shows the type descriptors that JOTAI extracts for the function `sum`, in Figure 5 (b). The descriptors expose the structure of aggregate type `struct S`, and list the types used in the signature of function `sum`.

(a)
```
01 struct S {
02   int data;
03   char flag;
04 };
05 typedef struct S MyStruct;
06 void sum(MyStruct s, int* p, int n) {
07   int sum = 0;
08   if (s.flag == 's') {
09     for (int i = 0; i < n; i++) {
10       sum += p[i];
11     }
12   }
13   s.data = sum;
14 }
```

(b)
```
struct S
 | data int
 | flag char

function sum void
 | s struct S
 | p int*
 | n int
```

Figure 5. Given the function `sum` in part (a), JOTAI extracts the type descriptors in part (b) of the figure.

Type descriptors are extracted from candidate compilation units via a `clang` plug-in implemented with the "RecursiveASTVisitor". This `clang` class is used to traverse and extract information from the Abstract Syntax Tree of C programs. Once type descriptors are extracted, the process of generation of inputs no longer uses the target function. In other words, this function is analyzed only during the extraction of type descriptors. After this point, the function is treated as a black box.

3.2 The Constraint Generation Language

Type descriptors are used as inputs in the process of generating *constraints* for a program. Constraints can be thought as *refined types* [24]. In other words, instead of saying that the type of a variable is an integer, we say that this type is an integer larger than zero, for instance. Figure 6 shows the grammar of the constraint generation language. Constraints define essentially two properties over variables: `value` and `length`. The former applies to any variable; the latter only to variables of pointer types. The constraint language also defines a few "algorithmic skeletons", which we shall explain in Section 3.2.2. Example 3.2 provides examples of constraint-based refinements.

Example 3.2. Consider n and p in the type descriptors of Figure 5 (b). The type of n is an integer, and p has a pointer type. Constraints let us:

- Relate n's value with a constant, e.g., $\texttt{value}(n) > 0$.
- Relate the length of the memory region referenced by p with a constant, e.g., $\texttt{length}(p) > 10$.
- Relate the two variables, e.g., $\texttt{length}(p) > \texttt{value}(n)$.
- Constrain values within buffers, e.g., $\texttt{length}(p) == \texttt{value}(n) + 1, \texttt{value}(p[n]) == '\backslash 0'$

```
constraint  ::=  comp ( ',' comp )*

    comp    ::=  arith (== | != | > | < | >= | <=) arith
             |   skeleton

   arith    ::=  element (* | + | / | – | %) element
             |   element

skeleton    ::=  linked (<name> ',' <integer>)
             |   dLinked (<name> ',' <integer>)
             |   binTree (<name> ',' <integer> ',' <integer>)

 element    ::=  const | (value | length) '(' variable ')'

 variable   ::=  <name> ( '[' ( <integer> | <name> | '_' ) ']' )*
             |   <name > ( '.' <name> )*

    const   ::=  <integer> | <floating-point> | <char> | <string>
```

Figure 6. Examples of type descriptors that JOTAI extracts for `struct S` and function `sum`.

3.2.1 Generating Constraints. Users do not write constraints for each benchmark. This process is implemented in Python, as an extension of JOTAI's constraint generation module. The constraint generation module lets users specify existentials (e.g., "exists") or quantifiers (e.g., "for all") ranging over type descriptors. Thus, the space of constraints is searched by code that JOTAI's users write in Python, and the constraints themselves are generated in textual format, and then passed to JOTAI's input generator. Example 3.3 shows examples of three different constraint generators.

Example 3.3. Figure 7 shows four different constraint generation methods. These methods are implemented in Python. They receive a list of constraints, which must be augmented, plus a type descriptor. The type descriptor contains a list of variables that are scalars, pointers or aggregates. The terminals in Figure 6 are represented as Python classes, which can be combined in various ways to build complex constraints.

```
def intBounds(ctrs, type_desc):
  for name in type_desc.scalars:
    ctrs += Value(name, 100)

def zeroEnd(ctrs, type_desc):
  for n in type_desc.scalars:
    for p in type_desc.pointers:
      c0 = Value(n, Plus(n, 1))
      c1 = Length(p, Value(n))
      c2 = Value(Arr(p, Value(n)), '\0')
      ctrs += [c0, c1, c2]
      remove(type_desc, p)

def eqValLen(ctrs, type_desc):
  for n in type_desc.scalars:
    for p in type_desc.pointers:
      ctrs += Length(p, Value(n))
      remove(type_desc, p)

def bigArr(ctrs, type_desc):
  for name in type_desc.scalars:
    ctrs += Value(name, 255)
  for name in type_desc.pointers:
    ctrs += Length(name, 65025)
```

Figure 7. Examples of four constraint generation methods.

3.2.2 Algorithmic Skeletons. It is possible to use `length` and `value` constraints to define recursive data structures such as linked lists and trees. However, we found it easier to define a small library of algorithmic skeletons to specify data structures. Currently, we define three skeletons, which Figure 6 shows: `linked`, for linked lists; `dLinked` for doubly linked lists; and `binTree` for binary trees. These skeletons are chosen according to the type descriptor. If the descriptor contains one recursive reference, JOTAI tries to use `linked` to generate linked lists. If the descriptor contains two recursive references, then JOTAI can use either `dLinked` or `binTree` to generate data structures.

3.3 Filtering Out Undefined Behavior

There exist ANSI C programs that can be compiled by any compiler that conforms to the different C Standards, but whose runtime behavior is undefined. Quoting Hathhorn et al. [30]: "*The C11 standard mentions situations that lead to undefined behavior in 203 articles*". Among such situations, 77 involve aspects of the C language itself; i.e., are produced by the use of grammatical constructions of the language. Another 24 undefined behaviors are caused by omissions by the parser or the preprocessor. And there are 101 undefined behaviors caused by misuse of functions and variables from the language's standard library [17] (Appendix J).

The benchmarks that we produce are reconstructed from programs available in open-source repositories. Thus, undefined behaviors present in these programs are likely to persist in their benchmark versions. Additionally, our type reconstructor might introduce undefined behavior into programs that were originally correct, as Example 3.4 shows.

Example 3.4. Figure 8 shows a program whose types were reconstructed by PSYCHEC. PSYCHEC reconstructs `u64` and `s64` as "`int`". However, the former was originally declared as "`unsigned long`", and the latter as "`long`". In this case, the left shift in Line 04 in Figure 8 yields undefined behavior, because the shift count exceeds the width of the type that PSYCHEC has inferred. The program in Figure 8, when compiled with `gcc -O0`, `gcc -O1` and `clang -O0` outputs the same value. However, if compiled with `clang -O1` then it produces a different value when running on OSX 11.2.

3.3.1 AddressSanitizer and KCC. Detecting undefined behavior is not easy. Indeed, with the current technology presently available, it might be impossible. As pointed out by Memarian et al. [36]: "*The divergence among the de facto and ISO standards, the prose form, ambiguities, and complexities of the latter, and the lack of an integrated treatment of concurrency, all mean that the ISO standard is not at present providing a satisfactory definition of C as it is or should be.*" Thus, to filter out undefined behavior, we adopt a best-effort approach, based on a combination of two tools: ADDRESSSANITIZER [46] and KCC [30].

We compile every candidate execution unit with `clang` and `gcc`, using, in both cases, the following flags: "`-fsanitize = address, undefined, signed-integer-overflow -fno`

```
01 static int foo(s64 nblocks) {
02   s64 sz, m;
03   int l2sz;
04   m = ((u64) 1 << (64 - 1));
05   for (l2sz = 64; l2sz >= 0; l2sz--, m >>= 1) {
06     if (m & nblocks) {
07       break;
08     }
09   }
10   sz = (s64) 1 << l2sz;
11   if (sz < nblocks) {
12     l2sz += 1;
13   }
14   return (l2sz - L2MAXAG);
15 }
16
17 int main(int argc, char *argv[]) {
18   int benchRet = foo(255);
19   printf("%d\n", benchRet);
20   return 0;
21 }
```

Types inferred by Psyche-C:

```
int L2MAXAG = 0;

typedef int s64;
typedef int u64;
```

Original types:

```
int L2MAXAG = 32;

typedef long s64;
typedef unsigned long u64;
```

Figure 8. The left shift in Line 04 causes undefined behavior in this program.

`-sanitize-recover=all`". These flags invoke different extensions from AddressSanitizer [46]. However, even the programs that run until normal termination (exit code zero), and that output the same results with either `clang` or `gcc` can still exhibit undefined behavior, as Example 3.5 illustrates.

Example 3.5. Consider the following program:
`int main() {int i = 3; i = i++; return i;}`
This program runs until normal termination when compiled with the AddressSanitizer plugins. However, once compiled with Kcc, it stops with the error code `UB-EIO8`, i.e.: "*Unsequenced side effect on scalar object with side effect of same object*" (see C11's Section 6.5:2).

Thus, to further remove malformed benchmarks, the programs that pass through the AddressSanitizer sieve are then compiled with Hathhorn et al.'s Kcc. These programs execute with a timeout. Using a timeout is paramount with Kcc, for it slows down the target programs by a substantial margin. Our experience with Kcc has presented us with situations where programs that do not seem to sport any undefined behavior would loop forever. Nevertheless, Kcc detects more occurrences of undefined behaviors than Asan.

3.3.2 Freeing Memory. The `length` constraint, described in Section 3.2, allocates memory for pointers. The `-fsanitize=address` flag mentioned in Section 3.3.1 will stop programs that leak memory. Thus, allocated memory must be freed before the benchmark terminates. To ensure deallocation, Jotai's input generator uses a table of memory blocks to store every block of memory that the input generator creates. Before the benchmark terminates, the table is traversed, and blocks stored there are freed. This approach ensures deallocation of memory used in recursive data structures, like the ones created by algorithmic skeletons.

4 Evaluation

This section analyzes five research questions related to the benchmarks produced via the Jotai methodology.

4.1 The Benchmark Generation Rate

Jotai benchmarks are produced out of programs publicly available in open-source repositories. Thus, the number of possible "seeds" for benchmarks is virtually unbounded. However, most of the functions in these repositories will not yield valid benchmarks. This section analyzes the rate in which benchmarks can be produced, in order to answer Question 1.

Question 1 (RQ1). *What is the ratio of candidate functions to viable programs generated by the methodology in Fig. 1?*

Benchmarks: To answer Question 1, we apply the methodology from Figure 1 onto the GPL 3.0 benchmarks publicly available in the AnghaBench repository on August 7th, 2022 (http://cuda.dcc.ufmg.br/angha/home). At that time, the repository had 1,041,333 compilable C functions taken from 148 GitHub repositories, sorted by the number of stargazers. Out of this lot, 70,309 functions are *leaf routines*; that is, functions that do not invoke other functions. We only apply the Jotai methodology to the leaf routines, to ensure the absence of invisible instructions in the benchmarks.
Hardware: Intel i7-6700T with 7.6GB of RAM.
Software: Benchmarks are compiled with `clang` 15.0 plus AddressSanitizer and with Kcc 3.4.
Methodology: We apply six constraints on each leaf routine. Three constraints are the built-in skeletons **linked** (**lk**), **dLinked** (**dl**) and **binTree** (**bt**). The former is applied onto functions containing an argument whose type has a recursive reference; **dl** and **bt** are applied onto types that contain two recursive references. The last three constraints are:

- **int-bounds (ib):** for every scalar n: $\texttt{value}(n) = 100$.
- **big-arr-10x (bx):** for every scalar n: $\texttt{value}(n) = 10$; and for every pointer p: $\texttt{length}(v) = 100$.
- **big-arr (ba):** for every scalar n: $\texttt{value}(n) = 255$; and for every pointer p: $\texttt{length}(v) = 65,025$.

Discussion: Figure 9 shows the number of valid compilation units produced with AddressSanitizer and with Kcc. Out of 1,041,333 functions in AnghaBench, 70,309 were candidate compilation units, leading, in the end, to 36,223 benchmarks that could be successfully compiled and executed with Asan and Kcc. The success rate of Kcc depends on a timeout. If we compile the first 1,000 compilation units produced via **int-bounds** with Kcc, using a timeout of one second, then we obtain 937 valid programs. Increasing this timeout to 10 seconds adds one more program to this collection. Kcc failed to compile 17 programs; 46 other programs stopped with clear error messages referring to the C11 Standard.

	ib	ba	bx	lk	dl	bt	#f
Asan	40,062	39,448	39,884	192	69	97	41,995
Kcc	35,198	34,396	34,780	169	47	47	36,223

Figure 9. Number of valid execution units produced with the Asan and the Kcc sieves. Results are cumulative: a program that passes the Kcc sieve has also passed the Asan sieve. **#f** is the total number of benchmark files produced. Each file contains at least one and at most six different inputs.

4.2 Normality

The goal of this section is to demonstrate how Jotai can be used as a means to analyze and understand the dynamic behavior of programs. Several statistical tests (e.g.: the T-test, the Z-test, Pearson's Correlation, etc) assume that data comes from a normal distribution. The normal distribution typically emerges from the accumulation of effects produced by independent events [16]. Given that the different optimization levels of a compiler are formed by the combination of different optimizations, one could be tempted to believe that optimization speedups obey a normal distribution. This section evaluates this hypothesis.

Question 2 (RQ2)**.** *Does the speedup observed after the application of compiler optimizations follow a normal distribution?*

Benchmarks: We evaluate Question 2 onto three different collections of programs:

- **SameRes:** The 19,211 execution units produced via the **big-arr** constraint that return a primitive value (`int`, (`uint`, `float`, etc).
- **DynGr:** The 856 execution units produced with **big-arr** whose number of instructions executed is larger than the number of different instructions fetched. These are programs containing loops that run at least twice.
- **Spec:** The 43 programs from SPEC CPU2017 [10]. We use **Spec** to give the reader some perspective on how Jotai programs compare with this well-established benchmark suite.

Hardware: Intel i7-6700T with 7.6GB of RAM.
Software: Benchmarks are compiled with `clang` 15.0. We count instructions using Valgrind [41]'s CfgGrind [45].
Methodology: We define *speedup* as the rate of instructions executed by the benchmark once compiled with `clang -O0` and `clang -OX`, where `X` $\in \{1, 2, 3, s, z\}$[1]. We measure speedup using the number of instructions counted by CfgGrind, instead of using running time, because this metric is stable—running time is subject to much variation. Section 4.3 provides evidence that this methodology is sound.
Discussion: Figures 10, 11 and 12 summarize the results observed in this experiment[2]. Each figure shows a density (left) and a quantile-quantile (right) plot. The former highlights means and variances in the measured speedups. The latter compares the observed distribution with a normal distribution with similar parameters. The gray area in the QQ-plot delimits the area where normal data would be expected to exist. In all three cases, including Spec CPU2017, the Shapiro-Wilk Normality Test returns a p-value lower than 0.0001, indicating that the observed speedups are very unlikely to come from a normal distribution.

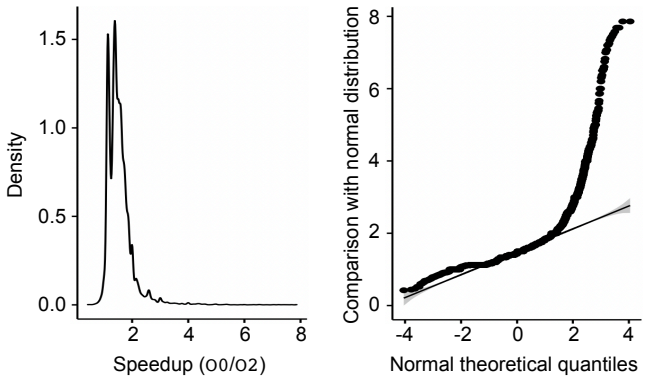

Figure 10. Density plot and QQ-plot for the 19,163 programs in the **SameRes** suite of benchmarks. The density plot is cropped at 8.0 to improve visualization. Mean = 1.54x, Median = 1.45x, Outliers (speedup greater than 8.0) = 130.

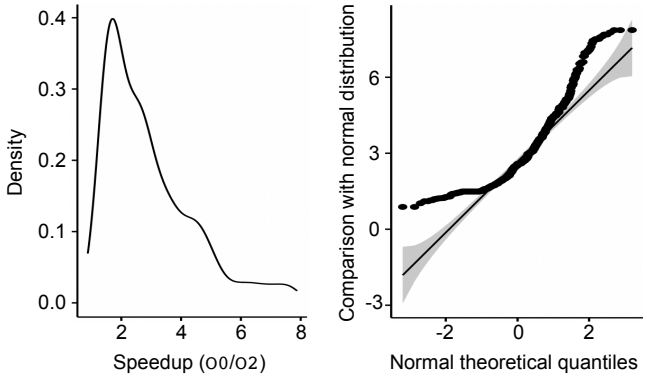

Figure 11. Density plot and QQ-plot for the 856 programs in **DynGr**. Mean = 2.91x, Median = 2.57x, Outliers = 48.

Optimizations bear stronger effects in programs containing loops, as the 90/10 Rule of Code Optimization implies [50, Ch.3]. Thus, speedups are higher on **DynGr** (Median = 2.57x) and **Spec** (Median = 2.72x) than on **SameRes** (Median = 1.45x). Nevertheless, even **SameRes** contains samples whose speedup would be impossible under a normal distribution. For instance, `clang -O2` provokes a speedup higher than 500x in two programs of **SameRes**. The probability of observing this speedup under the assumption of a normal distribution is zero for all practical purposes. Indeed, it is easy to

[1]To keep the presentation short, we plot speedups relative to `clang -O2`; however, we could observe very similar results for the other levels.
[2]To ease visualization, we crop the speedup at 8.0x in every density plot.

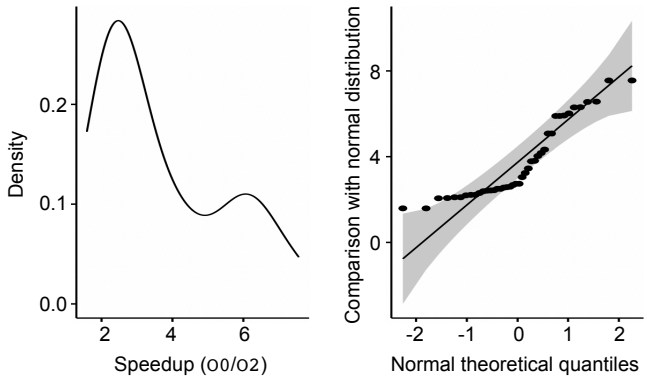

Figure 12. Density plot and QQ-plot for the 43 programs from SPEC17. Mean = 3.70x, Median = 2.72x, Outliers = 1.

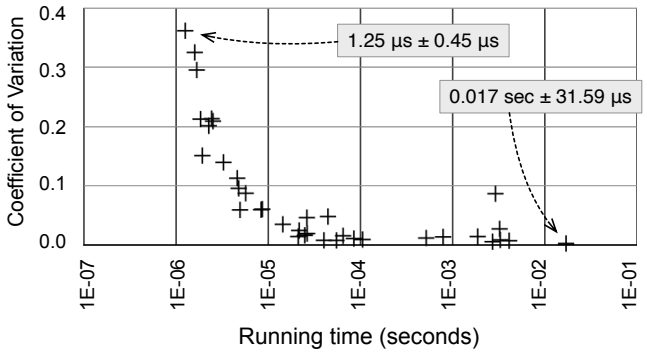

Figure 13. Coefficient of variation for the 50 programs that run more instructions when compiled with `clang -O0` in the Jotai collection.

write programs where the speedup obtained after standard optimizations can be as large as one wants. As an example, both `clang` and `gcc` are able to replace loops that sum arithmetic progressions with $O(1)$ formulae.

4.3 Running Time vs Instructions Fetched

Programs produced via the Jotai methodology tend to run for a very short time—they are code snippets. Fast execution complicates using these programs to tune compilers, because measurements become imprecise. Imprecisions manifest in terms of high coefficients of variations (the ratio between standard deviation and mean). As a consequence, differences in the running times of variations of the same benchmark cannot be distinguished via statistical tests, e.g, if the p-value is used as the statistic to perform a significance test, then the result will be inconclusive. Example 4.1 illustrates this issue.

Example 4.1. Figure 13 shows the coefficient of variation for the 46 programs from the Jotai collection that run the largest number of instructions when compiled with `clang -O2`. The mean running time of these programs (arithmetic averages of 10 samples) varies from 1.25 microseconds to 17 milliseconds. The coefficient of variation of the 5 fastest programs is always above 0.3 (i.e., 30%) and always below 0.03 (i.e., 3%) for the 5 slowest programs.

Although the running time of Jotai benchmarks can vary, the number of instructions that they execute is fixed: the programs are deterministic and only contain visible instructions. This observation motivates the research question that this section explores:

Question 3 (RQ3). *How strong is the correlation between the number of instructions executed by a Jotai benchmark and its running time?*

Benchmarks: The 46 programs seen in Example 4.1[3].

[3]Initially, we tried to use 50 programs; however, 4 of them could not be used with Valgrind in the Odroid board, due to excessive memory consumption. Without Valgrind they work correctly.

Hardware: We measure correlations in two processors:

x86: i7-6700T, at 2.80GHz, with 7.6GB of RAM
arm: Odroid XU4 A15, at 2.00GHz, with 2GB of RAM

Both processors contain eight cores; however, programs run sequentially, at maximum frequency.

Methodology: We average the time of 10 executions of each benchmark, compiled with `clang -O0` or `clang -O2`. We count the number of instructions fetched per benchmark using CfgGrind. Running time and instructions refer to the function that constitutes the benchmark—the rest of the driver is not analyzed.

Discussion: Figure 14 plots the running time of programs versus the number of instructions they fetch. In contrast to running time, the number of instructions executed is fixed per benchmark. Figure 14 uses log scale in both axes; hence, it gives the false impression that programs compiled at `-O0` fetch as many instructions as programs compiled with `clang -O2`. However, as already seen in Section 4.2, this difference is large. To emphasize this distance, Figure 15 summarizes all the populations displayed in Figure 14.

Figure 16 shows the Spearman and the Kendall Rank Correlation Coefficients between the running time and the number of instructions executed per benchmark. Both ranks are non-parametric; hence, recommended in the analysis of data that comes from non-normal distributions. In both cases, correlation is very high, tending to 1.0. We have omitted Pearson's Coefficient, which is not robust in face of outliers. Nevertheless, Pearson's Coefficient is also greater than 0.9 in all four cases. Notice that although absolute times tend to be very different in the two boards, speedup ratios of `clang -O2` over `clang -O0`, considering averages, sums, maximums and minimums tend to be very similar.

4.4 Speedup Prediction

Modern compilers, such as Clang, provide different levels of optimization aimed at increasing the efficiency of executing code. The application of these optimizations can lead to a

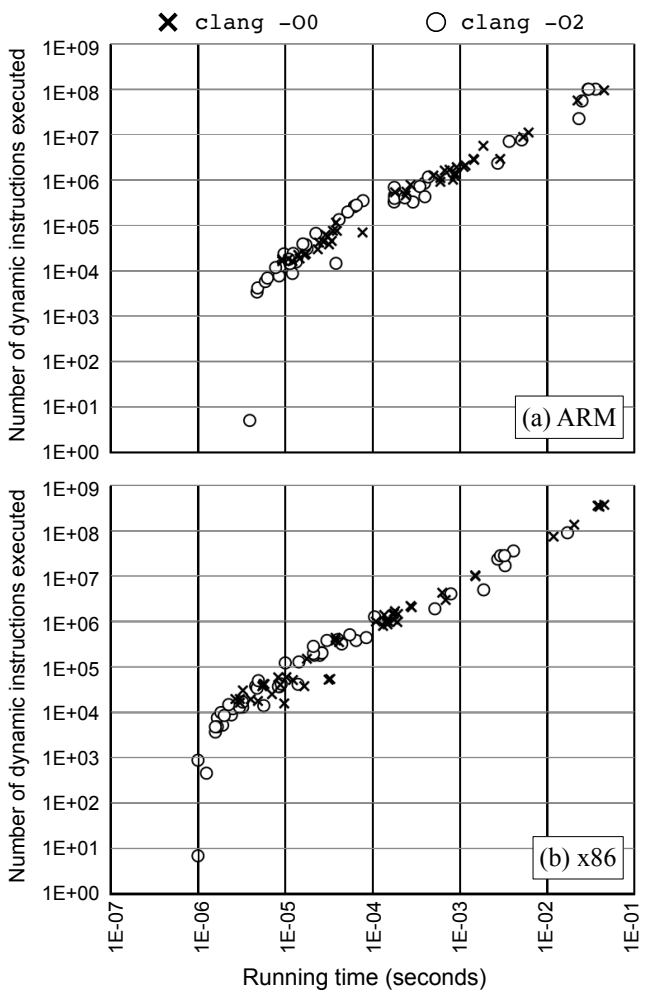

Figure 14. The correlation between the running time and the number of instructions executed per benchmark in two different architectures: (a) ARM and (b) x86, considering programs compiled at two different optimization levels.

		clang -O0		clang -O2	
		time (sec)	# instrs.	time (sec)	# instrs.
ARM A15 (2.00GHz)	Sum	1.0193	2.62E+09	0.1882	5.04E+08
	Avg	0.0222	5.69E+07	0.0041	1.09E+07
	Min	9.17E-06	16,372	3.92E-06	5
	Max	0.2322	5.99E+08	0.0359	1.00E+08
x86 i7 (2.80GHz)	Sum	0.1997	1.65E+09	0.0372	2.45E+08
	Avg	0.0048	3.93E+07	0.0009	5.83E+06
	Min	4.00E-06	15,304	1.00E-06	7
	Max	0.0448	3.66E+08	0.0171	9.26E+07

Figure 15. Summary of data used to produce Figure 14.

significant performance increase, commonly referred to as 'speedup'. However, the degree of this improvement can vary

	ARM A15 (2.00GHz)		x86 i7 (2.80GHz)	
	clang -O0	clang -O2	clang -O0	clang -O2
Spearman	0.9642	0.9876	0.9903	0.9052
Kendall	0.8584	0.9206	0.9333	0.9791

Figure 16. Non-parametric correlation ranks between running time and number of instructions executed for the benchmarks in Figure 14.

widely among different programs. In this section, we explore the use of histograms of features extracted from the source code as predictive indicators for speedup.

Question 4 (RQ4). *Can the analysis of source code attributes represented as histograms lead to accurate predictions of the speedup from compiler optimizations?*

Benchmarks: The collection of 35583 programs produced with **big-arr**.
Software: Instructions are counted using Valgrind [41]'s CfgGrind [45]. The predictive models utilize the standard linear regression implementation provided by the Keras library. Each program's individual features are captured into a histogram using a custom Clang plugin, with features representing opcode identifiers, loop depths, and the number of basic blocks.
Methodology: Two linear regression models, built using the Keras library, are employed in our analysis. The first model, `Model_O0`, exclusively uses the histograms collected from programs compiled with `clang -O0`. The second model, `Model_O0_O1`, incorporates histograms obtained from both `clang -O0` and `clang -O1` flags. The 'speedup' is defined as the ratio between the number of instructions executed by a program when compiled with `clang -O0` and when compiled with `clang -O1`. In order to assess the relative performance of the predictive models, the geometric mean of the speedups obtained from the dataset served as a baseline for comparison. The models were evaluated using Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics quantify the discrepancy between predicted and actual values, with lower values indicating superior model performance.

Model Version	MSE	MAE
Model_O0	0.00701	0.05353
Model_O0_O1	0.00592	0.04605
Geometric mean	0.01561	0.10137

Table 1. Error Metrics for Regression and Geometric Models

Discussion: Table 1 summarizes the results observed in this experiment. The findings indicate that the Keras models outperform the geometric model in terms of both error metrics, irrespective of the dataset used for training. Further, the decrease in MSE and MAE values in the `Model_O0_O1` compared to the `Model_O0` suggests that including histograms

compiled with the `clang -O1` option improves the model's prediction capacity.

The results suggest that histograms derived from opcode identifiers, loop depths, and the number of basic blocks in the source code can successfully predict speedup resulting from compiler optimizations. Histograms are straightforward data structures: they can be extracted from programs via one linear pass over the program's code. Thus, we speculate that histograms could be used, for instance, in the context of a Just-in-Time compiler, to gauge the profitability of compiling programs with different optimization levels.

4.5 Coverage

Most of the programs that Jotai produces are very simple: their execution amounts to a linear path of basic blocks. Nevertheless, some large programs can be found in this collection. This section provides the reader with some idea about structural properties of these programs, namely, the average number of basic blocks visited and the proportion of branches traversed during execution of benchmarks.

Question 5 (RQ5). *What is the expected size of Jotai benchmarks, and what is the portion of this size that can be covered by simple constraints?*

Benchmarks: The 856 programs in the **DynGr** collection described in Section 4.2.
Hardware: Intel i7-6700T, at 2.80GHz, with 7.6GB of RAM
Software: Same apparatus seen in Section 4.2.
Methodology: We use CfgGrind to count the number of basic blocks visited during the execution of each benchmark. CfgGrind reconstructs the *dynamic slice* of the program. In other words, it builds the control-flow graph formed by the instructions fetched during the execution of the program. Such a dynamic slice is formed by *basic blocks*, i.e., maximal sequences of instructions that can execute in sequence, and *phantom blocks*, i.e., targets of branches that have not been visited. If a dynamic slice does not contain phantom blocks, then it has been completely visited during the execution of the program.
Discussion: Figure 17 shows the number of basic blocks visited and the number of phantom blocks observed during the execution of the 856 benchmarks in **DynGr** using the **big-arr** constraints. We show data for benchmarks compiled with different optimization levels of `clang`. On average, benchmarks have four to six basic blocks, with median five. The median value of phantom blocks is zero for most optimization levels. The number of benchmarks that are fully covered (i.e., that do not contain phantom blocks) varies per optimization level, peaking at 564 with `clang -Oz`.

Figure 18 summarizes the data presented in Figure 17. The row **sum** is the total of basic blocks visited during the execution of the benchmarks at different optimization levels or the total of different phantom blocks encountered during execution. Optimizations tend to reduce the number of basic blocks; however, this behavior is not always true: `clang -O3` increases the number of basic blocks, due to control-flow replication. Code vectorization, for instance, might replicate the body of loops. Nevertheless, although the static size of the program grows, its dynamic size—the number of instructions fetched—tends to decrease, as seen in Section 4.2.

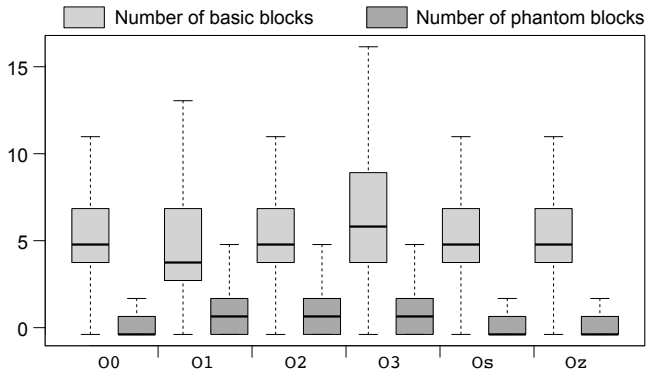

Figure 17. Number of basic blocks and phantom blocks found during the execution of the programs in **DynGr**.

		O0	O1	O2	O3	Oz	
basic	**median**	5	4	5	5	5	
	mean	5.999	5.143	5.359	6.764	5.395	
	max	24	20	17	117	19	
	sum	5,135	4,402	4,587	5,790	4,618	
phantom	**median**	0	1	1	1	0	
	mean	0.7465	1.027	0.9206	1.68	0.4579	
	max	15	6	6	20	5	
	sum	639	879	788	1,438	392	
	full	489	424	415	283	564	

Figure 18. Summary of the data used to produce Figure 17 (The 856 programs in the **DynGr** dataset introduced in Section 4.2). **Full** is the number of programs that did not contain phantom nodes, i.e., fetched branches with untaken paths.

5 Related Work

The development of compilers requires benchmarks. Thus, some of the most celebrated papers in programming languages describe benchmark suites, such as Spec CPU2006 [31], MiBench [28], Rodinia [11], etc. These benchmarks are manually curated, and typically comprise a small number of programs. Recently, Cummins et al. [14] have demonstrated that this reduced size fails to cover the space of program features that a compiler is likely to explore during its lifetime. Thus, researchers and enthusiasts have been working to generate a large number of diverse and expressive benchmarks. This section covers some of these efforts.

Random Synthesis. The generation of benchmarks for tuning predictive compilers has been an active field of research in the last ten years. Initial efforts directed to the development of predictive optimizers would use synthetic benchmarks conceived to find bugs in compilers. Examples of such synthesizers include CSmith [52], LDRGen [4] and Orange3 [39, 40]. Although conceived as test-case generators, these tools have also been used to improve the quality of the optimized code emitted by mainstream C compilers [5, 29]. However, more recent developments indicate that synthetic codes tend to reflect poorly the behavior of human-written programs; hence, yielding deficient training sets [18, 26].

Guided Synthesis. Several research groups have used guided approaches to synthesize benchmarks [6, 12, 14, 20]. These techniques might rely on a template of acceptable codes, like Deniz and Sen [20] do, or might use a machine-learning model to steer the generation of programs, like Cummins et al. [14] or Berezov et al. [6] do. Synthesis is restricted to a particular domain, like OpenCl kernels [14, 20]; or regular loops [6]. The approach described in this paper is different in the sense that the programs in Jotai are not synthesized; rather, they are mined from open-source repositories.

Code Mining. This paper produces benchmarks out of code from open-source repositories. We follow the methodology introduced by da Silva et al. [18] to extract and reconstruct programs, as Figure 1 illustrates. There exists a large body of literature about scraping programs from repositories. Some of these works aim at generating benchmarks to feed machine-learning models [23, 27]; however, to the best of our knowledge, only da Silva et al. [18] and Armengol-Estapé et al. [2] mine compilable programs to autotune compilers. Nevertheless, open-source repositories are not the only source of benchmarks. For instance, Richards *et al.* have produced realistic JavaScript benchmarks, out of monitored browser sections [44]. A shortcoming of Richards et al.'s approach is scalability: a human being still needs to create a browsing section that will give origin to one benchmark.

Generation of Executable Benchmarks. Many artificial benchmarks execute [2, 6, 14, 47]. However, except for Berezov et al. [6]—which generates specific-domain loops— these collections follow CLDrive's [14] approach to filter out incorrect kernels. In the words of Tsimpourlas et al. [47]: "*[CLDrive] rejects kernels that (i) produce runtime errors [observable crashes]; (ii) do not modify any of the inputs (no output) or (iii) modify them differently for each run (not deterministic)*". Notice that this approach still leaves room for undefined behavior. Indeed, all our attempts to run benchmarks produced by Berezov et al. and Armengol-Estapé et al. stumbled on undefined behaviors, which were reported (and confirmed) by these researchers. These previous generators also do not provide users with a way to explore the space of valid inputs, like the DSL that we introduce in Section 3.2— rather, the input generator is hardcoded into the synthesizer. As an example, Cummins et al.'s CLDrive uses only one approach to produce inputs, which is similar to the **big-arr** constraint described in Section 4.2.

6 Conclusion

This paper has introduced Jotai: a set of principles, techniques and tools to generate executable C benchmarks. Benchmarks consist of compilable C files containing, each, an executable function mined from an open-source repository. Compilation is achieved via type reconstruction. Sound execution is achieved via constraints defined in a domain-specific language to refine the type signature of functions. Programs in Jotai can be used in a variety of ways: from stress testing processors and compilers to autotuning compilation tasks. Currently, Jotai programs are distributed as part of a standalone repository, or as a CompilerGym dataset. We are already aware of user stories outside our group. For instance, Krister Walfridsson has used Jotai programs to test PySmtGcc, a translation-validator for Gcc. In his words, "*it detected cases that were missing in PySmtGcc* [48]."

References

[1] Andrei Rimsa Álvares, José Nelson Amaral, and Fernando Magno Quintão Pereira. 2021. Instruction visibility in SPEC CPU2017. *J. Comput. Lang.* 66 (2021), 101062. https://doi.org/10.1016/j.cola.2021.101062

[2] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. 2022. ExeBench: An ML-Scale Dataset of Executable C Functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3520312.3534867

[3] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive Modeling Methodology for Compiler Phase-Ordering. In *PARMA-DITAM*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/2872421.2872424

[4] 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

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

[6] Maksim Berezov, Corinne Ancourt, Justyna Zawalska, and Maryna Savchenko. 2022. COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks. In *PARMA-DITAM (Open Access Series in Informatics (OASIcs), Vol. 100)*, Francesca Palumbo, João Bispo, and Stefano Cherubin (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:14. https://doi.org/10.4230/OASIcs.PARMA-DITAM.2022.3

[7] Lars Bjertnes, Jacob O. Tørring, and Anne C. Elster. 2021. LS-CAT: A Large-Scale CUDA AutoTuning Dataset. *CoRR* abs/2103.14409 (2021), 6 pages. arXiv:2103.14409 https://arxiv.org/abs/2103.14409

[8] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2017. Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3. *CoRR* abs/1703.08228 (2017), 10 pages. arXiv:1703.08228

[9] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *CC*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.

3377894

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE* (Berlin, Germany). Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[11] 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

[12] Alton Chiu, Joseph Garvey, and Tarek S. Abdelrahman. 2015. Genesis: A Language for Generating Synthetic Training Programs for Machine Learning. In *CF* (Ischia, Italy). Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/2742854.2742883

[13] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*, Vol. 139. PMLR, Baltimore, Maryland, USA, 2244–2253.

[14] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, Piscataway, NJ, USA, 86–99. https://doi.org/10.1109/CGO.2017.7863731

[15] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. *CoRR* abs/2109.08267 (2021), 12 pages. arXiv:2109.08267

[16] S. J. Cyvin. 1964. Algorithm 226: Normal Distribution Function. *Commun. ACM* 7, 5 (may 1964), 295. https://doi.org/10.1145/364099.364315

[17] Comitê da Linguagem. 2004. Defect report #260. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm JTC 1, SC 22, WG 14.

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

[19] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. 2023. A Game-Based Framework to Compare Program Classifiers and Evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3579990.3580012

[20] Etem Deniz and Alper Sen. 2015. MINIME-GPU: Multicore Benchmark Synthesizer for GPUs. *ACM Trans. Archit. Code Optim.* 12, 4, Article 34 (nov 2015), 25 pages. https://doi.org/10.1145/2818693

[21] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *SP*. IEEE, Washington, DC, USA, 472–489. https://doi.org/10.1109/SP.2019.00003

[22] Anderson Faustino, Edson Borin, Fernando Magno Quintão Pereira, Otávio Nápoli, and Vanderson Rosário. 2021. New Optimization Sequences for Code-Size Reduction for the LLVM Compilation Infrastructure. In *SBLP*. ACM, New York, NY, USA, 33–40. https://doi.org/10.1145/3475061.3475085

[23] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-Free Probabilistic API Mining across GitHub. In *FSE* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

[24] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

[25] Grigori Fursin. 2014. Collective Tuning Initiative. *CoRR* abs/1407.3487 (2014), 27 pages. arXiv:1407.3487 http://arxiv.org/abs/1407.3487

[26] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. A Case Study on Machine Learning for Synthesizing Benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. ACM, New York, NY, USA, 38–46. https://doi.org/10.1145/3315508.3329976

[27] Georgios Gousios and Diomidis Spinellis. 2017. Mining Software Engineering Data from GitHub. In *ICSE-C* (Buenos Aires, Argentina). IEEE Press, Washington, DC, US, 501–502. https://doi.org/10.1109/ICSE-C.2017.164

[28] 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

[29] 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.

[30] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 336–345. https://doi.org/10.1145/2737924.2737979

[31] 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

[32] Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. 1999. A Feasibility Study in Iterative Compilation. In *Proceedings of the Second International Symposium on High Performance Computing (ISHPC '99)*. Springer-Verlag, Berlin, Heidelberg, 121–132.

[33] Amy McGovern and Eliot Moss. 1999. Scheduling Straight-Line Code Using Reinforcement Learning and Rollouts. In *NIPS*. MIT Press, Cambridge, MA, USA, 903–909. https://doi.org/10.5555/340534.340836

[34] 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

[35] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. 2020. Type Inference for C: Applications to the Static Analysis of Incomplete Programs. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 15 (2020), 71 pages. https://doi.org/10.1145/3421472

[36] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *PLDI* (Santa Barbara, CA, USA). Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[37] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. 1997. Learning to Schedule Straight-Line Code. In *NIPS*. MIT Press, Cambridge, MA, USA, 929–935. https://doi.org/10.5555/3008904.3009034

[38] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. AAAI Press, Phoenix, Arizona, 1287–1293. https://doi.org/10.5555/3015812.3016002

[39] 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.

[40] Kazuhiro Nakamura and Nagisa Ishiura. 2015. Introducing Loop Statements in Random Testing of C compilers Based on Expected Value Calculation. , 226–227 pages.

[41] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[42] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE]

[43] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Project CodeNet. https://doi.org/10.5281/zenodo.4814769

[44] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. *SIGPLAN Not.* 46, 10 (2011), 677–694.

[45] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *ATC* (Boston, MA). USENIX Association, USA, 28.

[47] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2022. BenchPress: A Deep Active Benchmark Generator. https://doi.org/10.48550/ARXIV.2208.06555

[48] Krister Walfridsson. 2022. Personal communication. Sent on October 29th.

[49] 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

[50] Bob Wescott. 2013. *Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With* (1st ed.). CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[51] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. 1998. Combining Loop Transformations Considering Caches and Scheduling. *Int. J. Parallel Program.* 26, 4 (aug 1998), 479–503.

[52] 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

[53] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *ICSE*. IEEE Press, New York, NY, US, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[54] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *ESEC/FSE* (Lake Buena Vista, FL, USA). ACM, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068