Liveness-Driven Random Program Generation

Gergö Barany

Liveness-Driven Random Program Generation

Gergö Barany

Inria, Paris, France
gergo.barany@inria.fr

Abstract. Randomly generated programs are popular for testing compilers and program analysis tools, with hundreds of bugs in real-world C compilers found by random testing. However, existing random program generators may generate large amounts of dead code (computations whose result is never used). This leaves relatively little code to exercise a target compiler's more complex optimizations.
To address this shortcoming, we introduce liveness-driven random program generation. In this approach the random program is constructed bottom-up, guided by a simultaneous structural data-flow analysis to ensure that the generator never generates dead code.
The algorithm is implemented as a plugin for the Frama-C framework. We evaluate it in comparison to Csmith, the standard random C program generator. Our tool generates programs that compile to more machine code with a more complex instruction mix.

Keywords: code generation, random testing, data-flow analysis, program optimization

1 Motivation

Optimizing compilers for real-world programming languages are complex pieces of software. Compiler bugs may manifest in several ways: As compiler crashes, missed optimizations, or as silent miscompilations. The third category is especially serious as it may introduce bugs in correct programs. Such wrong-code bugs may invalidate all correctness guarantees provided by source-level verification of safety-critical (and other) software systems.

Two main avenues of work address these problems: compiler verification and compiler testing. Compiler verification has seen much research [4], with CompCert as a prominent example [9]. However, such compilers have not entered the mainstream yet: Compiler verification is difficult and time-consuming, and verified compilers therefore perform fewer optimizations and target fewer CPU architectures than others.

A different approach is to test compilers in a way that instills confidence. Standard compiler test suites exist for exercising C compilers, in particular for testing their conformance to various details of the standard [15,16]. In addition, randomized differential testing has gained prominence in recent years. Compiling many random programs with various compilers and comparing the behaviors of

the generated binaries can uncover input programs that cause compiler crashes or miscompilations. The best-known example of this approach is the work of Yang et al. on Csmith [20], a generator of random C programs. Csmith generates programs that are fully self-contained (including all their inputs in initialized global variables) and conform to the C standard by construction. If two compilers produce code that behaves differently for a Csmith-generated program, one of the compilers must contain a miscompilation bug. Testing of C compilers with Csmith has uncovered hundreds of bugs in total, including crashes and miscompilations in every compiler under test. This included bugs in (unverified parts of) the CompCert verified C compiler [20].

This article describes a random generator of C programs developed for a project on finding missed optimizations in C compilers. Inspired by the successes of Csmith, in this project we generate random C programs, compile them using various compilers, then use custom tooling to search for possible optimizations in the resulting binaries. (The details are described in a separate paper [2].)

Starting with Csmith as our program generator, we found early on that it was not an optimal fit for our intended use case: Despite generating realistic-looking programs with complex arithmetic expressions, accesses to global and local variables including through pointers, structures, and arrays, as well as nested loops and branches, it produces large amounts of *dead code* whose results are never used. (See our experiments in Section 4.) Dead code elimination, a standard part of every optimizing compiler, can thus remove large parts of the code generated by Csmith, leaving very little relevant code for the remaining more interesting optimizations. Csmith often generate functions of several hundred lines of code that are compiled to ten machine instructions, representing only a small fraction of the computations present on the source code level.

In this paper we address this problem with our new *liveness-driven* random generator `ldrgen`. Our tool uses liveness analysis during program generation to avoid generating dead code. In the following sections we describe liveness-driven program generation; the implementation of our tool as a Frama-C plugin; and its experimental evaluation, showing that `ldrgen` generates programs that compile to a larger amount of code and a more complex instruction mix than programs generated by Csmith.

2 Fully Live Programs

In this section we briefly recall the basics of liveness analysis and then present our novel formulation as a set of structural inference rules.

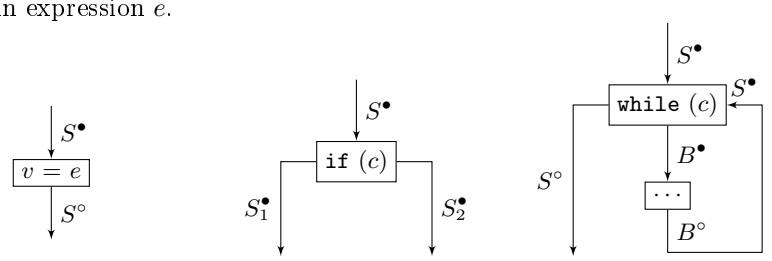
2.1 Principles of Liveness Analysis

A variable is called *live* at a program point if the value it holds at that point may be read in the future, without an intervening redefinition; otherwise, it is called *dead*. For example, in a code snippet like `x = a + b; x = 0; return x;`, the variable `x` is dead after the first assignment but live after the second one

(because it is used in the `return` statement). We can extend the notion of liveness from variables to the assignment statements defining them: An assignment v `=` e is live iff the variable v is live just after this statement. The first assignment to `x` above is dead, the second one is live. (Unfortunately, some authors use the term *dead code* to refer to *unreachable* code, as in `if (false) x = y`. These concepts are not the same; our use of the terms *live* and *dead* does not refer to reachability.)

Dead assignments without other side effects are useless and can be removed from the program. Even mildly optimizing compilers implement a dead code elimination pass that would completely remove the addition from the first program fragment above. Our goal is to generate only live code, i. e., only code that does *not* contain any such opportunities for dead code elimination.

Liveness analysis is one of the classical data-flow analyses [13]. It is a *backward*, *may* analysis traditionally performed as backward fixed-point iteration over a program's control-flow graph. A statement S in the control-flow graph has a *live-in* set $S^\bullet$ and a *live-out* set S° which capture the sets of live variables before and after execution of S. Every statement S also has a *transfer function* f_S relating these sets. Figure 1 shows the transfer functions for assignments, `if` statements, and `while` loops. Liveness information is noted on the edges of the control flow graph. In the equations, $FV(e)$ denotes the set of all variables in expression e.



$$S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e) \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c) \qquad S^\bullet = S^\circ \cup B^\bullet \cup FV(c)$$

Fig. 1. Liveness transfer functions

The transfer function for an assignment $v = e$ is said to *kill* the variable v and *generate* all the variables in e. The transfer functions for `if` statements and `while` loops kill nothing and generate the variables in their condition c. The live-in set $S^\bullet$ of a loop S has a recursive dependency on itself: It is the same as the body's live-out set B°, on which the body's live-in set $B^\bullet$ depends. The desired solution for $S^\bullet$ is the *least fixed point* of the system of equations, which can be found efficiently by propagating data-flow information backwards around the loop.

2.2 Recognizing Fully Live programs

We will call a program *fully live* if all of the assignment statements it contains are live. This section develops an inference system characterizing fully live programs.

Figure 2 shows the abstract syntax of our programming language of interest, a subset of C function bodies without declarations. The language contains variables, constants, and all side-effect-free arithmetic and bitwise operators of C. Statements are assignments (the only source of side effects), `return` statements, `if` statements and general `while` loops. In contrast to C's concrete syntax, we view the semicolon `;` as a statement separator, not a terminator. For now, the language does not include `for` loops, nor any structures, arrays, or pointers. All variables are considered local.

$v ::= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{x} \mid \texttt{y} \mid \ldots$	variables
$n ::= \texttt{0} \mid \texttt{1} \mid \ldots$	constant literals
$e ::= v \mid n \mid e \texttt{ + } e \mid e \texttt{ - } e \mid \ldots \mid e \texttt{ << } e \mid \ldots \mid \texttt{-}e \mid \texttt{!}e \mid \ldots$	expressions
$S ::= v \texttt{ = } e$	assignment statement
$\mid \texttt{return } v$	return statement
$\mid \texttt{\{\}}$	empty block
$\mid S; S$	sequencing
$\mid \texttt{if } (e)\ S \texttt{ else } S$	conditional branch
$\mid \texttt{while } (e)\ S$	loop

Fig. 2. Abstract syntax of a C-like programming language

Figure 3 shows a system of inference rules that characterize fully live programs. In these rules we use a notation similar to Hoare triples. A *liveness triple*

$$\langle S^\bullet \rangle \ S \ \langle S^\circ \rangle$$

means that the variables in the set $S^\bullet$ are live immediately before the statement S (*live in*), and the variables in S° are live immediately after S (*live out*). A program S is fully live iff there is a set of variables $S^\bullet$ such that the liveness triple $\langle S^\bullet \rangle \ S \ \langle \emptyset \rangle$ is derivable in the system.

Intuitively, the system of inference rules encodes two things. First, the rules are an alternative presentation of the transfer functions of liveness analysis. A triple $\langle S^\bullet \rangle \ S \ \langle S^\circ \rangle$ that appears in a valid derivation corresponds to a data-flow equation $S^\bullet = f_S(S^\circ)$ where f_S is the transfer function for the statement S. For example, the transfer function $f_{v=e}$ for an assignment v `=` e is just $f_{v=e}(S^\circ) = (S^\circ \setminus \{v\}) \cup FV(e)$, as in the side condition of the Assign rule. Similarly, the Sequence rule encodes the composition of transfer functions, and the If rule encodes the split and join of data-flow information along different program paths.

Second, the other side conditions add a system of constraints to ensure full liveness: Any assignment statement appearing in a fully live program S (i. e., for which a derivation of $\langle S^\bullet \rangle \ S \ \langle \emptyset \rangle$ for some $S^\bullet$ exists) is itself live. This follows directly from the Assign rule's side condition $v \in S^\circ$. For example, a triple of the form

$$\langle S^\bullet \rangle \texttt{ x = a; x = b } \langle S^\circ \rangle$$

$$\textsc{Return}\ \frac{}{\langle \{v\}\rangle\ \texttt{return}\ v\ \langle \emptyset \rangle} \qquad \textsc{Skip}\ \frac{}{\langle S\rangle\ \{\}\ \langle S\rangle}$$

$$\textsc{Assign}\ \frac{v \in S^\circ \qquad S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)}{\langle S^\bullet\rangle\ v = e\ \langle S^\circ\rangle}$$

$$\textsc{Sequence}\ \frac{\langle S_1^\bullet\rangle\ S_1\ \langle S_2^\bullet\rangle \qquad \langle S_2^\bullet\rangle\ S_2\ \langle S_2^\circ\rangle \qquad S_2^\bullet \neq \emptyset}{\langle S_1^\bullet\rangle\ S_1\ ;\ S_2\ \langle S_2^\circ\rangle}$$

$$\textsc{If}\ \frac{\langle S_1^\bullet\rangle\ S_1\ \langle S^\circ\rangle \qquad \langle S_2^\bullet\rangle\ S_2\ \langle S^\circ\rangle \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c) \qquad S_1 \neq \{\} \vee S_2 \neq \{\}}{\langle S^\bullet\rangle\ \texttt{if}\ (c)\ S_1\ \texttt{else}\ S_2\ \langle S^\circ\rangle}$$

$$\textsc{While}\ \frac{\langle B^\bullet\rangle\ B\ \langle B^\circ\rangle \qquad B^\circ = S^\bullet\ (\text{minimal}) \qquad S^\bullet = S^\circ \cup B^\bullet \cup FV(c) \qquad S^\circ \neq \emptyset}{\langle S^\bullet\rangle\ \texttt{while}\ (c)\ B\ \langle S^\circ\rangle}$$

Fig. 3. System of inference rules for fully live programs

can never be derived in the system because the first of the two assignments is dead. The SEQUENCE rule says that to derive this triple, there must be some intermediate set S' of variables such that $\texttt{x} \in S'$ due to ASSIGN on `x = a` while at the same time $S' = (S \setminus \{\texttt{x}\}) \cup \{\texttt{b}\}$ due to ASSIGN on `x = b`. This is a contradiction, so the derivation attempt must fail.

While the other rules are straight-forward, the WHILE rule deserves some discussion. Unlike the two branches of the `if` statement, the whole loop's live-out set S° is not identical to the loop body's live-out set B°: Typically there are loop-carried dependences, i. e., cases where a variable is set on one iteration of the loop and its value is read on a later iteration. Such variables are live out of (and live into) the loop body, but if they are no longer used once the loop has terminated, they are not live out of the loop. When performing a derivation in the system, we must guess or calculate the set of these additional variables.

Let f_S denote the liveness transfer function corresponding to the loop body statement S. Then from the liveness triple $\langle B^\bullet\rangle\ S\ \langle B^\circ\rangle$ we have $B^\bullet = f_S(B^\circ)$, and the equation $B^\circ = S^\circ \cup B^\bullet \cup FV(c)$ means that B° is a fixed point of the function $\lambda B.\ (S^\circ \cup f_S(B) \cup FV(c))$. The minimality side condition additionally specifies that we are interested in the *least* fixed point of this function. This least fixed point exists and is unique [13].

In Figure 4 we illustrate the use of the inference system to prove full liveness of a program taking an input variable `n` (assumed to be non-negative) and returning the `n`-th Fibonacci number. We omit some details to focus on the analysis of the loop. Note that only the return variable `a` is live after the loop. However, the live-out set of the loop's body is $\{\texttt{a}, \texttt{b}, \texttt{n}\}$. This includes the return variable `a` and

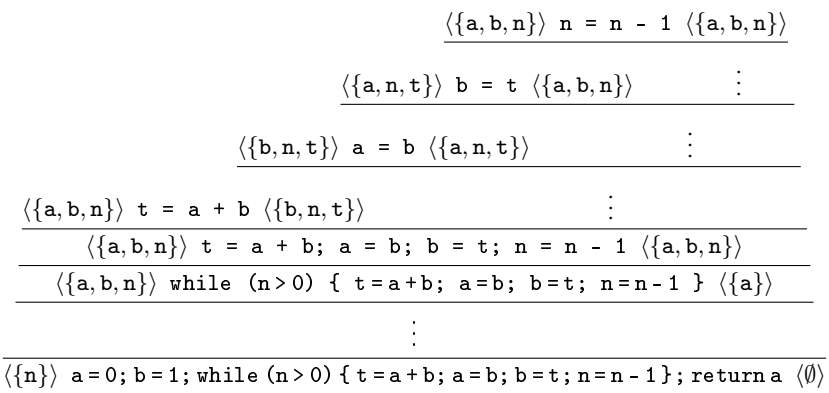


Fig. 4. Example derivation proving full liveness.

the variable `n` that is used in the loop condition. It also includes the variable `b`, which is the element computed by fixed point iteration: The value of `b` at the end of the loop body will be used on the next loop iteration, if any. Conversely, if `b` were not live at some point in the loop body, our inference system would not allow derivation of a triple for the assignment `b = t`. Indeed, all assignments in the loop body satisfy the condition that they define variables that are live after the assignment. That is, this program is *fully live* by our definition.

2.3 Limitations of the System

Note that fully live programs may still contain opportunities for simple optimizations that remove code that does not have interesting effects. For example, programs accepted by the inference system above may contain fragments like `if (0) { ... } else { ... }` where one of the branches of the `if` statement is unreachable and thus irrelevant; or assignments like `y = ...; x = y - y;` where the computation for the value of `y` is irrelevant for `x`'s final value of 0. Our inference rules do not consider the semantics of the code in enough detail to exclude such cases.

Our claims with regards to full liveness are relative to a purely syntactic notion of liveness that does not consider such semantic issues. In particular, we cannot guarantee that the liveness analysis embedded in these rules is equivalent to liveness analysis as performed by any given compiler. Any other analyses or transformations performed by the compiler before liveness analysis may influence the results, typically making the compiler's results more precise than ours.

As our experimental results in Section 4 show, our generator performs well nonetheless, so we can leave refinements of the system for future work.

2.4 Generating Fully Live Programs

The inference rules can be translated almost directly into an executable random (or exhaustive) generator of fully live programs. Like traditional liveness analysis, generation proceeds backwards, i. e., in the direction opposite control flow.

The side conditions of the inference rules ensure that a fully live program always ends in a return statement, as no other statement may have an empty live-out set. The generator can thus start by picking a random program variable v and generating a statement `return` v with live-in set $L = \{v\}$. It then iteratively prepends random statements S to the current program fragment and updates the live-in set according to $f_S(L)$. The possibilities for the generation of S are guided by L. In particular, if the generator decides to generate an assignment statement, the target variable v must be an element of L at that point. Conversely, if L ever becomes empty, generation of the current block of code must stop at that point: Any code preceding that point would be dead. Figure 5 shows pseudocode of such a generator in an OCaml-like functional language.

Every statement generation function takes a live variable set L (representing the live-out set S° of the statement S to be generated) and returns a pair of a newly generated statement and an updated live variable set according to the statement's transfer function. We iterate statement generation and collect a list of statements forming a block. In this presentation we omit helper functions such as the ones for generating random variables and expressions.

As before, the handling of loops merits more discussion. Just as the inference system needs a minimal set $B^\bullet$ containing new variables that are live into and out of the loop body, the random generator must synthesize such a set of variables. But here the problem is more difficult: During inference, we can start with an initial live-out set and derive the eventual live-in set by fixed-point iteration. During code generation this is not possible since we cannot analyze the loop body before we have constructed it. Instead, we first generate a random set of newly live variables and let this choice guide generation of the loop's code.

This solution relies on the following observation: The new variables we are interested in are ones that are defined before the loop, may be defined on some loop iteration, and then used on some later iteration. In the example of Figure 4, this is the case for variable `b`, which stores the next Fibonacci number for assignment to `a` on the subsequent iteration. Using the names in the WHILE rule of Figure 3, the set of these 'new' variables is $B' = B^\circ \setminus (S^\circ \cup FV(c))$. It follows that $B' \subseteq B^\bullet$, i. e., every $b \in B'$ is live into the loop.

To generate a fully live loop, we choose a random set B' of new variables and generate a loop body in a way that ensures that $B^\circ = S^\circ \cup B' \cup FV(c)$ is a least fixed point of the loop. For this we add B' to the live variable set L before generating a loop body block (along with the live variables $FV(c)$ generated by the loop condition). The generated loop body may or may not define or use variables in B', and thus these variables may or may not be live into the generated block. However, we must force them to be used in the loop body and be live into the body: If there were some b in the generated B' that is not live into the loop body, we would violate the condition $B' \subseteq B^\bullet$ established above.

```
let random_statements L code =
    if L ≠ ∅ then
        let (S, L') = random_statement L;
        random_statements L' (S :: code)
    else
        (code, L)

let random_statement L =
    let statement_generator = random_select [assignment; branch; loop];
    statement_generator L

let assignment L =
    let v = random_select L;
    let e = random_expression ();
    ("v = e", (L \ {v}) ∪ FV(e))

let branch L =
    let (t, L1) = random_statements L [];
    let (f, L2) = random_statements L [];
    let c = random_expression ();
    ("if (c) t else f", L1 ∪ L2 ∪ FV(c))

let loop L =
    (* See main text for explanation. *)
    let c = random_expression ();
    let B' = random_variable_set ();
    let (code, L') = random_statements (L ∪ B' ∪ FV(c)) [];
    let V = {b ∈ B' | b ∉ L' or b not used in S};
    if V ≠ ∅ then
        let e = random_expression_on_variables V;
        let v = random_select L';
        let code' = "v = e" :: code;
        ("while (c) code'", (L' \ {v}) ∪ V ∪ L)
    else
        ("while (c) code", L' ∪ L)

(* Start generation with the terminating return statement. *)
let v = random_variable ()
let (code, L) = random_statements {v} ["return v"]
```

Fig. 5. Pseudocode of a liveness-driven random program generator.

On the other hand, if b were live into the loop body but did not have a use anywhere in the body, then $B^\bullet$ would not be minimal and hence B° would not be a least fixed point of the constraint system.

To ensure a correct, minimal solution, we therefore find the set V of all $b \in B'$ that are not in L' or that are in L' but have no use in the generated loop body. We pick a random live variable $v \in L'$ and prepend an assignment `v = e` to the generated loop body, where e is an expression containing all the variables in V. This final loop body ensures that all variables in B' are live into it and used in it, hence ensuring that $B^\circ = S^\circ \cup B' \cup FV(c)$ is a least fixed point of the loop's liveness constraint system.

A small detail not illustrated in the pseudocode is the case when L' is empty at the beginning of the generated loop body. This can only be the case if the first statement in the body is an assignment of a constant expression (i. e., not using any variables) to some variable v, since such assignments are the only statements that can remove variables from the live variable set without adding any new ones. In this case, we replace this assignment's right-hand side with the expression e generated as above.

3 Implementation

We have implemented the random program generation algorithm sketched above as a plugin for Frama-C [7]. Frama-C is a general, extensible framework for source-level analysis and transformation of C programs. It is written in OCaml and can be extended with plugins written in that language. For this work, we do not need any advanced Frama-C features but benefit from its AST (abstract syntax tree) type definitions, utilities for managing variables and constructing AST fragments, and its pretty-printer for outputting the generated AST as C source code. As these general parts are provided by Frama-C, `ldrgen` itself can be quite small: It consists of only about 600 lines of generator code, plus some utilities and configuration.

`ldrgen` is free software, available at `https://github.com/gergo-/ldrgen`.

3.1 Random Generation

The core of the generator has the same structure as the pseudocode in Figure 5. After generating an empty function definition and a return statement, it fills in the function's body by generating a fully live sequence of statements as in the pseudocode. Statements are represented by AST fragments; we never need to worry about generating actual C syntax. The current version of `ldrgen` always generates a single function.

Random expressions are generated by choosing an operator among the arithmetic operators available in C and recursively generating the appropriate number of operand expressions. At this point, C's type system becomes relevant; if needed, we insert type casts to ensure all operands of an operator have the same type. Type casts are also needed in some other cases: Bitwise operators and the

modulo operator cannot be applied to floating-point numbers in C, so we insert conversions to integer types in such cases.

Many C operators may invoke undefined behavior when applied to inappropriate values. Two examples are division by zero and signed integer arithmetic overflow. Unlike Csmith in its default mode, `ldrgen` does not try to guard against such undefined operations, except for two cases that compilers have repeatedly warned us about: We clamp the right-hand-side operands of bit-shift operations to the bit size of the expression on the left-hand-side, and we always generate division and modulo operations of the form $e_1/(e_2+c)$ for some constant c instead of just e_1/e_2. The idea behind this is that e_2+c is less likely to evaluate to zero than a random expression in general. This approach is primitive, but we have found it to work well in practice.

Leaves of expressions are constants or variables. For constant literals we simply generate a random number. For a variable use we either use a previously used variable or generate a new one. Variables generated in this way may be local variables or function parameters. Both can be used in expressions, but we only generate assignments to locals, not to parameters.

Some of the generator's choices are weighted by manually chosen parameters to ensure generation of somewhat more realistic-looking programs. For example, we prefer generation of basic arithmetic operations to bitwise operators. We also ensure that loop and branch conditions are not constant expressions, i. e., that they contain at least one variable. In order to avoid trivial non-termination issues, we also ensure that every loop body's final statement is an assignment to some variable that occurs in the loop condition. If there were no modification of any of these variables at all, a loop once entered could never terminate. Even so, termination is not at all guaranteed.

Bottom-up generation of the function's body may stop if there are no more live variables, or if a user-defined limit is reached. In this latter case, there may remain live local variables at the start of the function's body. Their liveness means that they may be used without being assigned to, so we must ensure that they are initialized. We therefore finalize the function definition by initializing all such live-in variables to constants or to the values of function parameters.

3.2 Configuration

The generator's behavior may be tuned using command-line arguments. These may specify features of the sub-language of C that is used. For example, the user may request the generation of code that only uses integer types, or only floating-point types. They may also specify that no bitwise operations or no divisions should be generated, and whether loops may be generated. Other flags specify structural properties: The maximal number of statements per block, and the maximal nesting depths of statements and expressions.

`ldrgen`'s random generation uses OCaml's standard pseudorandom number generator, which can be seeded with a random seed or with a seed value specified as a command line argument. Invoking a given version of `ldrgen` with a fixed set of arguments and a fixed seed thus always gives the same reproducible result.

3.3 Extensions to the Basic Model

We describe two extensions to the core language of Figure 2 that are already implemented in `ldrgen`: very limited uses of pointers and `for` loops over arrays.

First, in addition to the arithmetic types used so far, we can generate function parameters of type `T *` (pointer to `T`) for some arithmetic type `T`. A parameter `p` of such type can be used in generated code as `*p`. We currently do not generate assignments to such dereferenced pointers, nor any pointer arithmetic.

Second, we want to generate arrays and restricted forms of loops over them in order to exercise loop optimizations such as unrolling or vectorization. For this we generate pointer arguments `T *arr` which are only used in `for` loops of the following form:

```
v = ...;
for (unsigned int i = 0; i < N; i++) {
    v = v ∘ f(arr[i]);
}
```

Here, `N` is a global variable considered to hold the array's size, f(`arr[i]`) is a random expression involving `arr[i]`, and $\circ$ is a randomly chosen binary arithmetic operator. This loop pattern implements a map-reduce operation, mapping some function f over the array and reducing (folding) the result with $\circ$. It is currently the only kind of `for` loop we implement, but this would be easy to generalize.

Similar forms of loops are already generated by Csmith, but there their results are virtually never used. Csmith's loops are therefore completely eliminated by compilers instead of being vectorized. In `ldrgen`, we choose a loop result variable `v` that is live after the loop to ensure that it is used, and the loop exposed to the compiler's loop optimizer.

3.4 Future Extensions

In the future, we are planning to extend `ldrgen` to generate structure types and allow the use of their members.

In the longer term, `ldrgen` will also be extended to support programs consisting of several random functions which may call each other. We are not planning to support non-structured control flow using `goto`. The more structured `break` and `continue` statements might eventually be supported, but this is not a priority as they complicate the structural liveness analysis.

4 Evaluation

The design goal of `ldrgen` was to have a random program generator that exposes as much interesting code as possible to all passes of the compiler under test; recall that we found Csmith-generated code to contain much dead code which is never seen by many parts of the compiler because it can be optimized away early on.

We will have achieved our goal if, for comparable amounts of generated C code, `ldrgen`'s output results in more, and ideally more varied, assembly code than Csmith's output. We therefore compare the two generators along these lines. We do not claim superiority to Csmith in any other regard, especially not concerning its power to find subtle miscompilation bugs. Csmith covers a larger subset of the C language than `ldrgen` and, in its default mode, carefully ensures that its output is well-defined according to the C standard. `ldrgen` tries to guard against oversized shifts and divisions by zero (see Section 3.1) but does not otherwise guarantee well-definedness.

Csmith is designed to run complete, self-contained applications consisting of several functions, driven by a `main` function. In contrast, `ldrgen` only generates individual functions without a driver. However, Csmith's many configuration options allow us to ask it to generate files consisting only a single function without `main`.[1]

Table 1 presents our experimental results for 1000 programs each generated by Csmith and `ldrgen`. We investigate three characteristics of the generated programs: lines of C code, number of instructions in the generated code, and number of unique opcodes in the generated code. In all cases, the C code was compiled to x86-64 machine code using GCC 5.4.0 with optimization setting `-O3`. For each characteristic, the table shows the total over the 1000 files as well as the minimum, median, and maximum values. (In cases where the median is not unique, we chose the arithmetic mean of the two closest values.)

Table 1. Comparison of code generated by Csmith and `ldrgen` in 1000 runs each.

	generator	min	median	max	total
lines of code	Csmith	25	368.5	2953	459021
	`ldrgen`	12	411.5	1003	389939
instructions	Csmith	1	15.0	1006	45606
	`ldrgen`	1	952.5	4420	1063503
unique opcodes	Csmith	1	8	74	146
	`ldrgen`	1	95	124	204

Our command line flags for Csmith were chosen in order to generate comparable numbers of lines of C code to `ldrgen`. In fact Table 1 shows that it generates somewhat more, but these numbers are difficult to compare precisely because Csmith-generated code tends to contain many initializers for global variables; `ldrgen` does not generate any global variables at all. We believe that the settings we chose allow a fair comparison of the generators.

Next we compare the number of instructions (executable code only, excluding static data, assembler directives etc.) emitted by the compiler for the generated source files. `ldrgen` was designed to increase this number compared to Csmith,

[1] The concrete flags we used were `-nomain -float -max-funcs 1 -no-safe-math -max-block-size 8 -concise`.

and the table shows that we have succeeded: While on average Csmith's output compiles to a single machine instruction per ten lines of code, `ldrgen`'s output has almost three instructions per single line of source code. Overall, `ldrgen`-generated programs compile to about 20 times as much machine code as Csmith-generated programs of comparable size. We can also see that the distribution for Csmith is highly skewed: The median shows that at least half of the functions generated by Csmith compile to 15 instructions or fewer. This also confirms our initial, more informal observation that Csmith-generated code tends to contain large amounts of dead code. `ldrgen` manages to generate code with a less skewed distribution, and in particular with generally higher numbers of emitted instructions.

On a side note, we remark that both Csmith and `ldrgen` sometimes generate functions that compile to a single machine instruction. Inspection showed that this happens in cases where the compiler recognizes that a function ends up in an infinite loop without externally visible side effects. Such functions are then compiled into a single unconditional jump instruction looping back to itself. Many other functions compile to two instructions, typically some simple operation on a function argument or a constant followed by a return. It would be difficult to completely avoid generating infinite loops, but comparatively easy (at least within `ldrgen`) to avoid generating functions that return after a single operation. For both Csmith and `ldrgen`, about 10 % of all cases fall into one of these trivial categories (with Csmith producing fewer infinite loops).

We analyze the coverage of the instruction set in the generated code by looking at the number of different opcodes generated. Here, too, we see that individual functions generated by `ldrgen` have a more varied instruction mix than functions generated by Csmith: Even the median for `ldrgen` is higher than the maximum for Csmith. Totaling over all the machine code in 1000 functions, we see that Csmith-generated code compiles to a mix of 146 different opcodes, while `ldrgen`-generated code contains 204 different opcodes, an increase in instruction set coverage of 40 %. Inspection of the sets of opcodes shows that this difference is almost entirely due to various vector (SIMD) arithmetic instructions generated for `ldrgen`'s code. Compiling to such instructions was the goal of adding `for` loops over arrays to `ldrgen`. Manual inspection of some cases shows that such loops are indeed the origin of these instructions. As noted above, such loops are also generated by Csmith, but their results are almost never used, so they do not appear in the compiled code. Disabling generation of `for` loops in `ldrgen` brings its total number of unique instructions down to 147, comparable to Csmith.

One of the few opcodes emitted for Csmith-generated code but not for `ldrgen` are `call` instructions to `memcpy` which are sometimes generated by compilers for structure copies. `ldrgen` currently does not generate structures at all.

Finally, in Table 2 we compare the speed of the two generators. Generating the 1000 files each analyzed above took 871 seconds with Csmith and 124 seconds with `ldrgen` (Csmith backtracks if it finds that it has generated unsafe code). Csmith generates about 527 lines of C code per second, with `ldrgen` generating 3140 (about 6 $\times$ more). With respect to final machine code, Csmith-

generated code compiles to about 52 instructions per second of generation time, whereas `ldrgen` produces 8563 (about 160 ×). These numbers do not include the time taken by the compiler; compiling all 1000 files for each generator takes about 46.6 seconds for Csmith and 80.3 seconds for `ldrgen`.

Table 2. Comparison of the time to generate 1000 files.

generator	time (sec)	lines/sec	instrs/sec
Csmith	871	527	52.4
ldrgen	124	3140	8562.8

5 Related Work

5.1 Csmith

The best-known random program generator is Csmith [20], based on an earlier system called `randprog` [5,19]. Csmith generates complete, self-contained programs that take all their input from initialized global variables and compute an output consisting of a hash over the values of all global variables at the end of execution. The generator is designed to only generate programs with well-defined semantics: Operations that may be undefined in C, such as overflowing signed integer arithmetic, are guarded by conditionals that exclude undefined cases (these guards can be disabled, and we disabled them for the experiments reported above). Like `ldrgen`, Csmith performs data-flow analysis during generation, although the details differ due to the differing design goals. Csmith's forward analysis computes points-to facts and uses them for safety checks. If the checks fail, Csmith backtracks, deleting code it generated until a safe state is reached again. In contrast, `ldrgen`'s data-flow analysis only deals with liveness, and `ldrgen` never backtracks: Full liveness of variables in loops is ensured by construction. Csmith generates a larger subset of C than current or currently planned versions of `ldrgen`, including unstructured control flow and less restricted use of pointers.

Due to Csmith's forward generation and data-flow analysis, it does not appear possible to directly integrate our backward liveness-driven approach in Csmith. A 'best effort' approach that would not give full liveness guarantees might extend Csmith with a forwards reaching definitions analysis [13]. At each program point, when generating an expression, Csmith could then prefer to use previously defined variables over fresh variables. Alternatively, it might be extended with a mode that computes a function's return value from all otherwise unused variables in the program, artificially making them live.

Csmith has been used to find hundreds of bugs in C compilers when compiling the programs it generates [20]. It has also been used as the basis of mutation-based systems, where Csmith's output was modified using other tools to provoke

compiler bugs [8]. The CLsmith tool derived from Csmith has been used to find many bugs in OpenCL compilers [10].

5.2 Other Random Generators

Other notable random generators include Orange3 [12] for C, jsfunfuzz [17] for JavaScript, and efftester [11] for OCaml. Our generator following a system of inference rules is similar to a generator based on Haskell's typing rules [14].

The JTT program generator [21] is aimed at testing compiler optimizations. It uses a model-based approach, with generation guided by test scripts. These scripts contain code templates and temporal logic specifications of the optimizations to be tested. For example, the authors specify opportunities for dead code elimination as cases where a variable is assigned, then assigned again before being used. The test script contains a temporal logic formula expressing this pattern and the test condition that the compiler should eliminate the first assignment. Using this script, JTT generates test programs containing this pattern.

Other work specifically aimed at testing and comparing program verification tools generates code from randomly generated LTL formulae [18]. The generated code is guaranteed to satisfy the specified temporal properties.

5.3 Structural Data-Flow Analysis

Our formulation of liveness analysis as set of structural inference rules is inspired by formulations of interval-based data-flow analysis where reducible programs are decomposed into components called *intervals*, and analysis data is efficiently propagated among the intervals [1,3,6].

6 Conclusions

We presented `ldrgen`, a new generator of random C programs designed for testing C compilers. In contrast to Csmith, the dominant player in this field, `ldrgen` is driven by liveness analysis to avoid generating dead code. We designed an inference system to capture our liveness analysis and implemented its rules as an excutable program generation system.

`ldrgen` is implemented as a plugin for the Frama-C framework. Our evaluation of `ldrgen` in comparison to Csmith shows that we have achieved our goal of generating C code that compiles to larger amounts of machine code with a more interesting instruction mix, including many SIMD instructions. We are actively using `ldrgen` in a project on finding missed optimizations in compilers [2]. Because it is able to exercise loop optimizations not usually addressed by Csmith, it may also be useful for finding correctness bugs in these optimizations.

Acknowledgments. The author would like to thank the anonymous reviewers, John Regehr, and Gabriel Scherer for insightful comments on earlier versions of this paper. This research was partially supported by ITEA 3 project no. 14014, ASSUME.

References

1. Allen, F.E.: Control flow analysis. SIGPLAN Not. 5(7), 1–19 (Jul 1970), http://doi.acm.org/10.1145/390013.808479
2. Barany, G.: Finding Missed Compiler Optimizations by Differential Testing. In: 27th International Conference on Compiler Construction (2018), http://dx.doi.org/10.1145/3178372.3179521
3. Cocke, J.: Global common subexpression elimination. SIGPLAN Not. 5(7), 20–24 (Jul 1970), http://doi.acm.org/10.1145/390013.808480
4. Dave, M.A.: Compiler verification: A bibliography. SIGSOFT Softw. Eng. Notes 28(6) (Nov 2003), http://doi.acm.org/10.1145/966221.966235
5. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: EMSOFT '08. ACM (2008), http://doi.acm.org/10.1145/1450058.1450093
6. Graham, S.L., Wegman, M.: A fast and usually linear algorithm for global flow analysis. J. ACM 23(1), 172–202 (Jan 1976)
7. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Comp. 27(3), 573–609 (2015)
8. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI '14. ACM (2014), http://doi.acm.org/10.1145/2594291.2594334
9. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (Jul 2009), http://doi.acm.org/10.1145/1538788.1538814
10. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. In: PLDI '15. pp. 65–76. ACM (2015)
11. Midtgaard, J., Justesen, M.N., Kasting, P., Nielson, F., Nielson, H.R.: Effect-driven quickchecking of compilers. Proc. ACM Program. Lang. 1(ICFP) (Aug 2017), http://doi.acm.org/10.1145/3110259
12. Nagai, E., Hashimoto, A., Ishiura, N.: Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. IPSJ Transactions on System LSI Design Methodology 7, 91–100 (2014)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
14. Pałka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: 6th International Workshop on Automation of Software Test. AST '11 (2011), http://doi.acm.org/10.1145/1982595.1982615
15. Perennial, Inc.: ACVS ANSI/ISO/FIPS-160 C validation suite, http://www.peren.com/pages/acvs_set.htm
16. Plum Hall, Inc.: The Plum Hall validation suite for C, http://www.plumhall.com/stec.html
17. Ruderman, J.: jsfunfuzz (2015), https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz
18. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: synthesizing programs of realistic structure. International Journal on Software Tools for Technology Transfer 16(5), 465–479 (Oct 2014)
19. Turner, B.: Random C program generator (2007), https://sites.google.com/site/brturn2/randomcprogramgenerator
20. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI '11. pp. 283–294. ACM (2011)
21. Zhao, C., Xue, Y., Tao, Q., Guo, L., Wang, Z.: Automated test program generation for an industrial optimizing compiler. In: ICSE Workshop on Automation of Software Test (2009)