Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages

Marcus Rodrigues
UFMG, Belo Horizonte, Brazil
maroar@dcc.ufmg.br

Breno Guimarães
UFMG, Belo Horizonte, Brazil
brenosfg@dcc.ufmg.br

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

***Abstract*—This paper presents a technique to generate in-bounds inputs for arrays used in memory-unsafe programming languages, such as C and C++. We show that most memory indexation found in actual C programs follows patterns that are easy to analyze statically. Based on this observation, we show how symbolic range analysis can be used to establish contracts between the arguments of a function and the arrays used within that function. To demonstrate the effectiveness of our ideas, we use them to implement Griffin-TG, a tool to stress-test C programs whose source code might be partially available. We show how Griffin-TG improves Aprof, a well-known algorithmic profiling tool, and we show how it lets us enrich PolyBench with a large set of new inputs.**

***Index Terms*—Test, Range Analysis, Static Analysis, Arrays**

I. Introduction

A programming language is termed *memory-unsafe* if it does not enforce in-bounds memory accesses. C, C++ and the vast majority of assembly dialects are memory-unsafe [1]. Although a source of bugs and vulnerabilities, this characteristic of programming languages is not an accident [2]. The need to enforce in-bounds memory accesses requires guards, and all the overhead they imply [3]. In the absence of such guards, programs can run faster, and type systems can be more permissive [4]. An evidence of the popularity that memory-unsafe languages enjoy is the long list of important systems implemented in them: browsers, virtual machines, operating systems and databases [5]. Such importance makes it hard to understand why we still lack techniques and tools to test software written in these languages.

A. Testing Memory-Unsafe Software is Hard

We shall use the program in Figure 1 to motivate this paper. Figure 1 shows a function, convol, that receives a square matrix mm of side N, and multiplies its row-th line by its col-th column. In this paper, we are interested in the following research question: how to generate inputs to test convol that respect the bounds of the arrays used within that function? There are several situations in which such ability is interesting, and in Section IV, we study two of them: to determine the complexity of convol empirically, and to infer its performance when compiled in two different ways.

Today there are tools that let us test convol automatically. Examples include KLEE [6], DART [7] and Austin [8]. However, none of them lets us answer our research question. To explain why they have shortcomings, we shall focus on KLEE, but the same arguments apply to the other software. KLEE is the most well-known symbolic execution tool currently available. To test convol using this tool, we can use the stub outlined in Figure 2. By running this stub multiple times, with different values for the command line arguments, we provide a pragmatic answer to our research question.

```
1  float* get_vector(int Width);
2
3  float* convol(float* mm, int row, int col, int N) {
4    int i, j;
5    float* v = get_vector(N);
6    for (i = 0; i < N; i++) {
7      v[i] = mm[(1+row) * (N+1) + i + 1] * mm[1 + col + (i+1) * (N+1)];
8    }
9    return v;
10 }
```

Fig. 1. The techniques proposed in this paper let us generate valid inputs for the convol function.

Nevertheless, we claim that this approach has two shortcomings, which this paper addresses. First, the code in Figure 2 was produced manually. Neither KLEE, nor Austin, nor DART are able to infer the relation between the size of vector mm and variables row, col and N. This relation, e.g., SIZE $=$ N $\times$ col $\times$ row is part of the *contract* of convol. Second, the stub in Figure 2 will only work if the body of function get_vector is present. In other words, KLEE will not generate a valid stub for this function; again, because it cannot infer the relation between N and the size of the vector v that get_vector returns. The methodology that we discuss in Section III-A addresses these limitations. Said methodology is not all-encompassing: it works only for code that shows certain properties. However, as we shall discuss in Section II, most real-world programs fit into this family.

B. Our Contributions

This paper describes a methodology to generate in-bounds inputs for arrays used in code written in memory-unsafe programming languages. Said methodology supports the automatic discovery of contracts to individual functions. Thus, we can test functions in the absence of the entire program where these functions belong, without ever producing inputs that lead to out-of-bounds memory accesses. To this end, we combine three different contributions:

978-1-7281-1436-1/19/$31.00 © 2019 IEEE

```
1  int main(int argc, char** argv) {
2    const int N = atoi(argv[1]);
3    const int row = atoi(argv[2]);
4    const int col = atoi(argv[3]);
5    const int SIZE = sizeof(float) * (N+1) * (N+1);
6    float *mm = (float*) malloc(SIZE);
7    klee_make_symbolic(mm, SIZE, "mm");
8    convol(mm, row, col, N);
9    return 0;
10 }
```

Fig. 2. Stub written to test function convol (Fig. 1) using KLEE. This paper lets us produce such stub automatically.

- **Insight:** in Section II we show that over 90% of memory accesses in real-world C programs are indexed by a family of symbolic expressions that depend only on *traceable* inputs that a function receives. We call these expressions *symbolic summations*.
- **Contracts:** in Sections III-A and III-B, we show how to use parameterized range analysis to bind memory ranges to program symbols. Our techniques work in tandem with any classic symbolic range analysis.
- **Resolution:** in Section III-C we explain how we can find valid solutions for symbolic summations. These solutions let us abide by the contracts that we have found through symbolic range analysis.

These contributions let us infer *dependent types* for arrays. Thus, instead of typing an array as int*, we type it as int[N×M], i.e., a vector with $M \times N$ cells. Contrary to previous work that infer dependent types for imperative arrays [9], we do it without the support of program annotations. Furthermore, by focusing on a family of indexing expressions, the symbolic summations, we can satisfy contracts without having to resort to combinatorial solvers, as previous work did [10]. Furthermore, our inputs lead to only in-bounds accesses, so our approach does not require tools like AddressSanitizer [11] or Dr. Memory [12] to ensure correctness of tests.

C. Results

We have materialized our ideas into a tool, the Griffin Test Generator (Griffin-TG). It receives as input a function written in C, infers a contract to it, and generates inputs that obey said contract. To demonstrate how Griffin-TG works, we use it in two scenarios. First, in Section IV-A, we show how Griffin-TG lets us circumvent a well-known problem in performance analysis [13]: the lack of enough inputs for benchmarks. We generate inputs for the PolyBench suite, and use them to compare different GPGPU runtime environments. Had we used only the standard data available in PolyBench, our conclusions would be different. Second, in Section IV-B we show how Griffin-TG improves Aprof, an algorithmic profiler [14]. To be effective, Aprof requires that a function be called many times with different inputs during program execution. This requirement is hard to enforce, because most functions are invoked only once [15]. Griffin-TG mitigates this problem, as it can generate an unbounded number of valid inputs to the same function. To demonstrate this point, we use Griffin-TG+Aprof to infer the complexity of popular sorting algorithms collected in the web.

II. Symbolic Summations

We want to generate inputs to test arrays used in functions, independent on the program where those functions exist, so that said inputs do not cause memory errors. The need to avoid out-of-bounds errors forces us upon two options:

- **Option 1**: generate values for the expressions used to index arrays, and then generate arrays whose size is larger than these expressions.
- **Option 2**: fix the size of an array, then produce values less than this size for the expressions indexing it.

While the former alternative is tempting, given the existence of techniques to determine the convex hull of memory regions [16], [17], we chose the latter. The applications of this work, namely, the performance test of programs, which Section IV illustrates, require us to vary the size of arrays in a controlled way. Option 1 could force us to always produce arrays that are either too big or too small for our purposes. Furthermore, an observation about the typical indexing expressions seen in actual programs gives us a simple algorithm to implement option 2, which works for many cases in practice. This observation is the subject of this section.

Any diophantine equation can be used to index an array; thus, both options 1 and 2, previously mentioned, involve solving a well-known undecidable problem, i.e., the resolution of diophantine equations [18]. The example in Figure 3 illustrates this difficulty. Because there exists no general procedure to solve diophantine equations, we must limit the kind of expressions that we analyze. In this paper, we restrict ourselves to the so called *Symbolic Summations*:

```
1  void zeros(int* v, int a, int b, int c) {
2    int aa = a*a*a, bb = b*b*b, cc = c*c*c, i = 0;
3    while (aa + bb != cc) {
4      v[i++] = 0; aa *= a; bb *= b; cc *= c;
5    }
6    v[i] = 1;
7  }
```

Fig. 3. This program illustrates the difficulty in finding precise bounds to arrays, so as to ensure the absence of out-of-bounds memory accesses in test generation. Until relatively recent developments [19], we did not know that the loop in line 3 would never stop.

Definition 2.1 (Symbolic Summations): The expression $x_1 + x_2 + \ldots + x_n + c, c \in \mathbb{Z}$ is a symbolic summation if each $x_k, 1 \leq k \leq n$ is an integer expression following the grammar: $x ::= k \mid s \mid x \times x \mid x + x$, where $k \in \mathbb{Z}, k > 0$ and s is a *traceable function input* (TFI) (see Def. 2.3).

To define the notion of a Traceable Function Input (TFI), we need to recall the concepts of *data and control dependencies*, as defined by Ferrante *et al.* [20]. A variable v is data dependent on a variable u if u is used in an expression that

computes the value of v. Moreover, v is control dependent on u if u is used in a predicate that determines which value is assigned into v. Example 2.2 illustrates these notions, and Definition 2.3 uses them to formalize the concept of TFI.

Example 2.2 (Dependencies): The assignments $v = u + 1$, $v = f(u)$, or $v = *u$ create a data dependence from u to v. C's ternary operation: $v = u\ ?\ 0\ :\ 1$, or the loop: $\texttt{while}(u)\{v = v + 1;\}$ create a control dependence from u to v.

Definition 2.3 (Traceable Function Input – TFI): A variable v used within the scope of a function f is a *first-order* traceable function input if v is:

- the formal parameters of f, and/or
- unambiguous global variables, and/or
- the return value of function calls.

A variable v is an n^{th}*-order* TFI if its computation only depends on TFIs of order at most n-1.

Definition 2.3 tells us that only variables loaded from ambiguous memory cells are untraceable. An ambiguous memory cell can be dereferenced by two or more pointers. There exists a rich literature on points-to analysis to find a conservative estimation of ambiguous memory regions [21], [22]. In this paper, for simplicity, we assume that the result of any *non-constant* load operation is ambiguous. A load operation is said to be constant if its target address cannot vary during program execution. Pointers allocated statically, e.g., globals and variables declared with the static modifier in C are constant, as long as their addresses are not taken. Aliasing precludes us from analyzing address-taken variables.

```
1  int* p1; int  g;
2  int get(int); int* new_array(int);
3  int foo(int i0, int* p0) {
4    int* p2 = new_array(i0 * sizeof(int));
5
6    int i1 = p2[0];      int i2 = i0 + g;    int i3 = i2 - i0;
7    int i4 = get(i3);    int i5 = i0 / i2;   int i6 = i2 * g;
8    int i7 = i0 * i2;
9
10   int i8; int i9; int i10 = i0; int i11 = i0;
11   for (i8 = 0; i8 < g; i8++) i10 += 2;
12   for (i9 = 0; i9 < i1; i9++) i11 += 2;
13
14   return p0[i0] + p0[i1] + p0[i2] + p1[i3] + p1[i4] + p2[i5]
15   + p2[i6] + p2[i7] + p2[i8] + p2[i9] + p2[i10] + p2[i11];
16 }
```

Fig. 4. The techniques in this paper can place correct bounds on the memory accesses marked in grey.

Example 2.4: Figure 4 shows examples of symbolic summations. Variable i1 is untraceable because it is loaded from a non-constant pointer. Variables i3 and i5 use operations forbidden in symbolic summations. Varibles i9 and i11 are control-dependent on i1; hence, untraceable.

The techniques discussed in Section III generate test inputs that never cause out-of-bounds accesses in symbolic summations, as long as we can use symbolic range analysis to find intervals for TFIs of order 2 or higher. Static range analysis is the subject of Section III-A. Such ideas are useful only if symbolic summations are common in programs. In the rest of this section, we provide some evidence to support this last statement. To this end, we have chosen two collections of programs to analyze: the LLVM test suite, and the GNU Coreutils library.

LLVM Test Suite. This collection contains, as of December of 2017, 518 programs, including well-known benchmarks such as FreeBench, McCat, MiBench, Prolangs and mediabench. In total, these benchmarks give us 2,905,560 assembly instructions and 663,902 memory accesses to analyze. Out of this lot, 612,769 indices are symbolic summations; that is to say, 92% of them. Figure 5 illustrates this relation for the 200 largest benchmarks in the LLVM test suite.

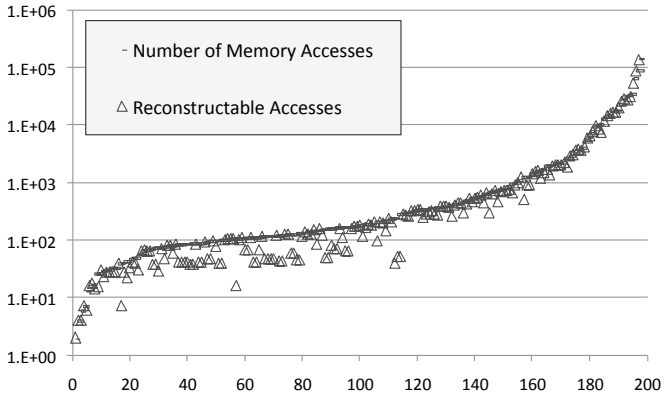

Fig. 5. Proportion of symbolic summations (reconstructable accesses) for the 200 largest benchmarks in the LLVM test suite. Ticks in the X-axis represent benchmarks. Benchmarks are sorted by number of memory access operations they contain. The Y-axis gives absolute numbers. Reconstructable memory accesses are memory indices that we can bound with the techniques seen in this paper.

GNU CoreUtils. This library contains C programs used in Unix-like operating systems, such as cd, cp, and ls. In contrast to LLVM's test suite, compiling them requires some manual work; hence, we shall analyze only the 10 first applications from GNU CoreUtils v8.27 – the latest version at the time of this writing. For each application, we have compiled the file containing its main routine. To make compilation possible, we had to reconstruct missing type definitions with Psyche-c [23]. The compilation of chown forced us to compile chowncore. Figure 6 summarizes our results. We found 5,090 memory accesses in these programs, and over 90% of them are symbolic summations. Most of the exceptions are due to loads from non-constant pointers. Only 3 memory accesses are non-reconstructable due to non-monotonic operations: one subtraction and two divistions.

The input generation technique discussed in this paper works for a function if all its memory accesses are indexed by symbolic summations. Figure 6 reports how many such functions we find in GNU Core Utils. Out of 54 functions, 39 contain just reconstructable memory indices. Only in two cases intractable accesses are due to non-monotonic indexing

Prog	Inst	M	TM	SM	F	TF	SF
base64	412	15	6	15	5	3	5
basename	197	9	6	9	4	3	4
chgrp	278	32	31	32	3	2	3
chroot	533	24	22	24	5	4	5
chmod	551	64	62	64	6	5	6
cksum	235	7	4	5	3	1	2
cat	678	44	42	43	6	4	5
chown	301	45	44	45	2	1	2
comm	751	246	212	246	5	2	5
chcon	507	30	29	30	6	5	6
chowncore	647	118	118	118	9	9	9
Total	5,090	634	576	631	54	39	52

Fig. 6. Occurrence of symbolic summations in GNU Core Utils. **Inst**: number of instructions in the program. **M**: number of memory accesses. **TM**: accesses indexed by traceable symbolic summations. **SM**: accesses indexed by symbolic summations (notice that **TM** $\subseteq$ **SM**). **F**: number of functions. **TF**: functions containing only accesses indexed by traceable symbolic summations. **SF**: functions containing only accesses indexed by symbolic summations.

expressions. Thus, the use of alias analyses would likely bring this number, 39, closer to the ideal 54.

Why Symbolic Summations? In this paper, we chose to generate inputs for symbolic summations because they are composed by *monotonic* operations. The absence of non-monotonic operators, such as division, and, most importantly, subtraction, gives us the opportunity to design an algorithm (Fig. 11) to find concrete replacements to these expressions that are always safe. The algorithm that performs this search and replacement is the subject of the next section.

III. Generation of In-Bound Inputs

Given a program P, our techniques produce a stub S that generates inputs for P. These inputs are guaranteed to honor the bounds of memory regions allocated in P. We create S in four steps. First, in Section III-A, we show how to calculate symbolic bounds to integer variables. Second, in Section III-B, we explain how these symbolic bounds are used to remove second-(or higher)-order TFIs from symbolic summations. Third, in Section III-C, we show how to "solve" these expressions formed only by first-order TFIs, meaning that, given a symbolic summation with bounded variables, we produce concrete values that respect said bounds. Finally, in Section III-D, we show how we use this ability to synthesize stubs.

A. Symbolic Range Analysis

The goal of Symbolic Range Analysis is to associate each integer variable v in a program with a *Symbolic Interval* $R(v) = [l, u]$, where l and u are *Symbolic Expressions*. A symbolic expression E is defined by the grammar below, where s is a *program symbol*[1] and $z \in \mathbb{Z}$:

$$E ::= z \mid s \mid E + E \mid E \times E \mid \min(E, E) \mid \max(E, E) \mid -\infty \mid +\infty$$

[1]For brevity, we shall call a "First-Order Traceable Function Input" (see Def. 2.3), a *Program Symbol*.

Any solution to symbolic range analysis must satisfy the correctness criterion given in Definition 3.1.

Definition 3.1 (Correctness of Symbolic Range Analysis): Let σ be a concrete store of a program P, so that $\sigma(S, v) = z$ is the value of variable v after the execution of the statement $S \in P$. Let R be a solution to the symbolic range analysis problem at S, e.g., $\mathbf{SB}(S) = R$, as defined in Figure 7. R is *valid*, if $\sigma(S, v) \in R(v)$.

1) Source-Level Implementation: The literature describes several algorithms that solve symbolic range analyses [24], [25], [26], [27], [28]. Thus, we shall not dive into its subtleties, except for one detail. These previous works implement range analysis in some variation of a low-level three-address code, whereas we do it at C's source level. Thus, for completeness, Figure 7 presents some of the constraints that we use to solve range analysis. This algorithm is not a contribution of this paper. A fixed point for such constraints can be found using the techniques of Blume *et al.* [25] or Nazaré *et al.* [26]. Theorem 3.3, at the end of Section III-A2, shows that a solution to the constraints meets the criterion of Definition 3.1.

Example 3.2: Figure 8 shows the result of computing range analysis for a typical C program. The constraints of Figure 7 might associate different symbolic ranges with the same variable at different program points. In this example, variable i is associated with different ranges at statements ℓ_1, ℓ_2 and ℓ_3. Notice that the ranges of program symbols are represented by the symbols themselves. For instance, the range of argc is $R(\text{argc}) = [\text{argc}, \text{argc}]$.

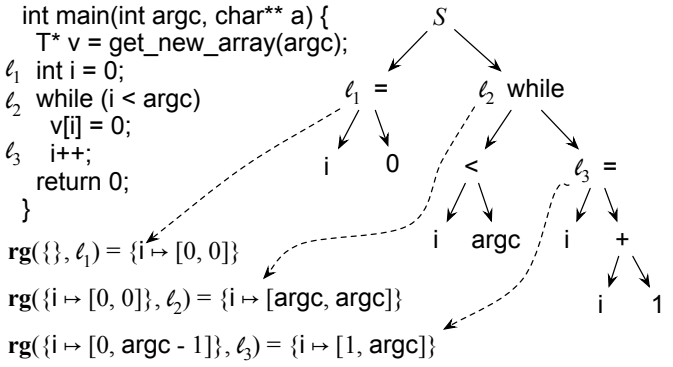

Fig. 8. Example of symbolic range analysis. We use labels $\ell_i, 1 \leq i \leq 3$ to denote points where we compute ranges.

2) Correctness of Range Analysis: The range analysis that we sketch in Figure 7 is a staple in the abstract interpretation literature. Nevertheless, for completeness, we show why this analysis is correct, and why we can use it to model more precisely the types of arrays. To prove that the abstract interpretation in Figure 7 correctly approximate the ranges of variables in a C program, we first define the semantics of a subset of this language. Figure 9 shows this semantics. An evaluation **ev** of a statement S of a program transforms a *store* σ into a new store σ'. A store is a map of variables to concrete values. For the sake of simplicity, we restrict these values

$$\mathbf{rg}(R, \mathsf{skip}) = R \qquad \frac{R' = R \setminus v \mapsto [n,n]}{\mathbf{rg}(R, v = n) = R'} \qquad \frac{R' = R \setminus v \mapsto [s,s]}{\mathbf{rg}(R, v = s) = R'} \qquad \frac{R(v_1) = [l,u] \qquad R' = R \setminus v \mapsto [l,u]}{\mathbf{rg}(R, v = v_1) = R'}$$

$$\frac{R(v_1) = [l_1,u_1] \qquad R(v_2) = [l_2,u_2] \qquad R' = R \setminus v \mapsto ([l_1 + l_2, u_1 + u_2])}{\mathbf{rg}(R, v = v_1 + v_2) = R'} \qquad \frac{\mathbf{rg}(R, S_1) = R_1 \qquad \mathbf{rg}(R_1, S_2) = R_2}{\mathbf{rg}(R, S_1; S_2) = R_2}$$

$$\frac{\begin{array}{c} R(v_a) = [l_a,u_a] \qquad R(v_b) = [l_b,u_b] \qquad R_t = (R \setminus v_a \to [l_a, \min(u_b - 1, u_a)]) \setminus v_b \to [\max(l_a + 1, l_b), u_b] \\ \mathbf{rg}(R_t, S_t) = R'_t \qquad R_f = (R \setminus v_a \to [\max(l_a, l_b), u_a]) \setminus v_b \to [l_b, \min(u_a, u_b)] \qquad \mathbf{rg}(R_f, S_f) = R'_f \qquad R' = R'_t \sqcup R'_f \end{array}}{\mathbf{rg}(R, \mathsf{if}(v_a < v_b)\ S_t\ \mathsf{else}\ S_f) = R'}$$

$$\frac{\mathbf{fp}(R, \mathsf{if}(v_a < v_b)\ S\ \mathsf{else\ skip}) = R'}{\mathbf{rg}(R, \mathsf{while}(v_a < v_b)\ S) = R'} \qquad \frac{\mathbf{rg}(R,S) = R}{\mathbf{fp}(R,S) = R} \qquad \frac{\mathbf{rg}(R,S) = R_1 \qquad R_1 \neq R \qquad \mathbf{fp}(R_1,S) = R'}{\mathbf{fp}(R,S) = R'} \qquad \frac{\mathbf{fp}(R,S) = R}{\mathbf{SB}(S) = R}$$

Fig. 7. Constraints produced for a few syntactic constructs of C for the symbolic range analysis. The relation $\mathbf{rg}(R, S)$ receives a command S, plus a function $R : V \mapsto [E, E]$ mapping variables to symbolic ranges, and produces a new binding of variables to ranges. The final range of variables *after* a statement S, denoted $\mathbf{SB}(S)$, is the fixed point (**fp**) of the relation $\mathbf{rg}(R, S)$. Notation $R \setminus v \mapsto [l, u]$ denotes a new function $\lambda\, x.(x = v)?[l, u] : R(x)$, and $R_1 \sqcup R_2$ denotes a function $\lambda\, x.R_1(x) \sqcup R_2(x)$.

to be integers. Given this semantics, Definition 3.1 states a correctness relation that our abstract interpretation preserves.

$$\frac{\sigma' = \sigma \setminus v \mapsto n}{\mathbf{ev}(\sigma, v = n) \Downarrow \sigma'} \qquad \frac{\sigma' = \sigma \setminus v \mapsto \mathsf{random}}{\mathbf{ev}(\sigma, v = s) \Downarrow \sigma'}$$

$$\frac{\sigma(v_1) = n_1 \qquad \sigma' = \sigma \setminus v \mapsto n_1}{\mathbf{ev}(\sigma, v = v_1) \Downarrow \sigma'}$$

$$\frac{\sigma(v_1) = n_1 \qquad \sigma(v_2) = n_2 \qquad \sigma' = \sigma \setminus v \mapsto (n_1 + n_2)}{\mathbf{ev}(\sigma, v = v_1 + v_2) \Downarrow \sigma'}$$

$$\mathbf{ev}(\sigma, \mathsf{skip}) \Downarrow \sigma \qquad \frac{\mathbf{ev}(\sigma_1, S_1) \Downarrow \sigma_2 \qquad \mathbf{ev}(\sigma_2, S_2) \Downarrow \sigma_3}{\mathbf{ev}(\sigma_1, S_1; S_2) \Downarrow \sigma_3}$$

$$\frac{\sigma(v_a) = n_a \qquad \sigma(v_b) = n_b \qquad n_a < n_b \qquad \mathbf{ev}(\sigma, S_t) \Downarrow \sigma'}{\mathbf{ev}(R, \mathsf{if}(v_a < v_b)\ S_t\ \mathsf{else}\ S_f) \Downarrow \sigma'}$$

$$\frac{\sigma(v_a) = n_a \qquad \sigma(v_b) = n_b \qquad n_a \geq n_b \qquad \mathbf{ev}(\sigma, S_f) \Downarrow \sigma'}{\mathbf{ev}(R, \mathsf{if}(v_a < v_b)\ S_t\ \mathsf{else}\ S_f) \Downarrow \sigma'}$$

$$\frac{\mathbf{ev}(\sigma, \mathsf{while}(v_a < v_b)\ S) \Downarrow \sigma'}{\mathbf{ev}(\sigma, \mathsf{if}(v_a < v_b)\ S; \mathsf{while}(v_a < v_b)\ S\ \mathsf{else\ skip}) \Downarrow \sigma'}$$

Fig. 9. Semantics of the subset of C's statements used to illustrate our symbolic range analysis.

Definition 3.1 leads to Theorem 3.3, which states that our symbolic range analysis is correct with regards to the correctness criterion. In other words, given a statement S, the relation **rg** produces an abstract stores S_R that correctly models any concrete store σ_S that could be reached by the evaluation of S, given an initial store that maps every variable to an unknown value, e.g., $\mathbf{ev}(\lambda x.\bot, S)$. In the proof of Theorem 3.3, particular attention must be given to symbols. A symbol s can have any value; hence, its range, $[s, s]$, can include any value, e.g., $n \in [s, s]$, for any n. In Figure 7's constraints, the initial abstract state of each variable is set to $\emptyset$; hence, it can only increase during abstract interpretation. To ensure that the analysis terminates, we use a widening

$$[l_1, u_1] \nabla_r [l_2, u_2] = [l, u], \text{ where } \begin{cases} l = l_1 & \text{if } l_1 = l_2 \\ l = -\infty & \text{otherwise} \\ u = u_1 & \text{if } u_1 = u_2 \\ u = +\infty & \text{otherwise} \end{cases}$$

Fig. 10. Widening operator used to ensure the convergence of the **rg** defined in Figure 7.

operator ∇_r, à la Cousout [29], which we define in Figure 10. Conservative widening ensures quick termination of our analysis, as any chain of intervals can only be stable towards $-\infty$ and $+\infty$.

Theorem 3.3: The symbolic range analysis of Figure 7 meets the correctness criterion of Definition 3.1.

Proof: The proof is by induction on the depth of the derivation tree produced by **ev**, when S is evaluated: We proceed by case analysis on each possible form S can have.

- S is skip: the result is immediate, as S has no variables, and the theorem is vacuously true.
- S is $v = n$: in this case, $R(v) = [n, n]$, and $\sigma(v) = n$, and we have that $n \in [n, n]$.
- S is $v = s$: we have that $R(v) = [s, s]$, where s is symbol. We also know that $\sigma(v) = \mathsf{random}$, and we have that $\mathsf{random} \in [s, s]$.
- S is $v = v_1 + v_2$. If v_1 is defined before being used, then by induction we know that $\sigma(v_1) \in R(v_1)$. Otherwise, v_1 is a symbol. In the latter case, we have that $R(v_1)$ includes any value. Same reasoning applies to $R(v_2)$. The result, e.g., $\sigma(v_1 + v_2) \in R(v_1 + v_2)$ follows from the definition of interval addition.
- S is $S_1; S_2$: The proof follows by induction. We apply induction on S_1, to infer that $R_1 \models \sigma_1$. From this conclusion, we find that $R_2 \models \sigma_2$.
- S is $\mathsf{if}(v_a < v_b)\ S_t\ \mathsf{else}\ S_f$: There are two possible evaluations of $v_a < v_b$. Case it is true, then we apply induction on S_t, plus the fact that $R(v_a)$ is upper bounded by the upper limit of $R(v_b)$. Similar reasoning holds if $v_a \geq v_b$.

- S is $\mathsf{while}(v_a < v_b)\ S$: This proof is the most complicated, because it requires us to know that the operator seen in Figure 10 is indeed a widening function. This is the classic widening operator used by Cousot and Cousot in the seminal paper about abstract interpretation [29]. There is a detailed proof of convergence in Nielson *et al.*'s book [30, Sec. 4.2.1]. Assuming that widening holds, then we have that the widen interval subsumes any interval before widening took place. Combining this fact with induction on S concludes the proof. □

B. Replacing Variables with Symbolic Bounds

According to Definition 2.3, a test generator can only control program symbols, i.e., first-order TFIs. Thus, to generate concrete values to a symbolic summation e used to index memory (e.g., $*(v+e)$ is in the program), we must discover the upper limit of e in terms of program symbols. As Theorem 3.4 states, the result of the symbolic range analysis contains only program symbols. Therefore, we can use this result to impose limits upon symbolic summations that contain only first-order TFIs.

Theorem 3.4 (Closeness): Let P be a program, and $R = \mathbf{SB}(S)$ be the result of symbolic range analysis at statement $S \in P$. For any variable $v \in S$, $R(v)$ is an expression formed by integer constants and program symbols.

Proof: the proof follows by induction on the derivation rules determined by the constraints in Figure 7. We show a few cases:

- $v = n$: the range $R(v) = [n, n]$ is formed by integer constants only;
- $v = s$, where s is a program symbol: the range $R(v) = [s, s]$ is formed by program symbols only;
- $v = v_1$: by induction, the range $R(v_1)$ is formed by program symbols or constants only, and we have that $R(v) = R(v_1)$;
- $\mathsf{if}(v_a < v_b)\ S_t\ \mathsf{else}\ S_f$: by induction, hypothesis holds for the ranges $R(v_a)$ and $R(v_b)$. The next ranges at S_t and S_f will include expressions min and max, and only the symbols in $R(v_a)$ and $R(v_b)$; □

To solve $e = i_1x_1 + \ldots + i_nx_n + c$, we replace each i_k with its upper limit, as determined by $R(i_k)$. Thus, if $R(i_k) = [l_k, u_k], 1 \leq k \leq n$, this step gives us a symbolic expression $u_1x_1 + \ldots + u_nx_n + c$. Because we are dealing with symbolic summations, we can disregard lower limits of variables when finding valid replacements for those expressions. Theorem 3.11 proves this statement. Furthermore, this substitution, when applied onto a symbolic summation, produces another symbolic summation, (Theorem 3.6).

Example 3.5: Range analysis gives us, at line 7 of Figure 1, the range $R(\mathsf{i}) = [0, \mathsf{N} - 1]$. To find upper bounds to the expressions that index mm, we can use the upper bound of this interval, i.e., $\mathsf{N} - 1$. Substitution on the first array index, e.g., $(1 + \mathsf{row}) \times (\mathsf{N} + 1) + \mathsf{i} + 1$ gives us the expression $(1+\mathsf{row}) \times (\mathsf{N}+1)+\mathsf{N}-1+1 = \mathsf{N} \times 2+\mathsf{N} \times \mathsf{row}+\mathsf{row}+1$. Similarly, substituting the range of i on the second memory access will give us $1 + \mathsf{col} + \mathsf{N} \times \mathsf{N} + \mathsf{N}$.

Theorem 3.6 (Preservation): If $e_i = i_1x_1+\ldots+i_nx_n+c$ is a symbolic summation, and $e_u = u_1x_1+\ldots+u_nx_n+c$ is the expression that results from the substitutions of upper bounds of each $i_k, 1 \leq k \leq n$, then e_u is still a symbolic summation, unless some $R(i_k) = [l, +\infty]$.

Proof: variables in a symbolic summation must be initialized with variables with positive coefficients, by definition. If they are created by decrement operations, e.g., $v = +v_1 - c$, then c must be an integer constant, and v's upper limit remains v_1's upper limit minus c. □

C. Solving Symbolic Summations

As discussed in Section II, to test programs, we first determine concrete values for the sizes of the arrays found in these programs; and then find valid concrete values for the indexing expressions. If a program contains a memory access $a[e]$, where e is a symbolic summation with the shape $u_1x_1 + \ldots + u_nx_n + c$, after replacement of upper bounds (Section III-B), then the size of a must be at least the constant part c. We denote this size by a_s+c, where a_s is an *open value*, i.e., a symbol that we can replace with concrete values later, when generating test inputs for a. Given any concrete input for a_s, we must ensure that $0 \leq u_1x_1+\ldots+u_nx_n \leq a_s$. The goal of this section is to explain how we find valid replacements for those symbols, so as to ensure in-bounds memory accesses. We shall call such replacements *realizations*.

Definition 3.7 (Realization): If e is an expression with k variables, then $n_s = e \setminus [v_i \mapsto n_i]^k$ is the concrete value of e after substituting each variable v_i with an integer value n_i. We call n_s the realization of this substitution. If $L = [(v_1, n_1), \ldots, (v_k, n_k)]$ is a list with k mappings $v \mapsto n$, then we let $e \setminus L$ denote the k substitutions $e \setminus [v_i \mapsto n_i]^k$.

To find a valid realization for an expression $e = u_1x_1 + \ldots + u_nx_n$, within the ranges $[0, a_s]$, we use the algorithm solve_all a_s L, where L is the list[2] $[u_1x_1, \ldots, u_nx_n]$. Figure 11 shows this algorithm. For conciseness, we present it in Standard ML/New Jersey (SML). This implementation is executable: one can try it as given. solve_all produces, as output, a list $[(v_1, n_1), \ldots, (v_x, n_x)]$. Each of these pairs denotes the fact that any concrete value of v_k must be less than or equal to $n_k, 1 \leq k \leq n$, to satisfy the original expression. To handle an expression e with n sums $e_k, 1 \leq k \leq n$, solve_all calls a recursive function solve on each e_k, giving it the n-th part of the allowed bound a_s. The datatype Exp denotes the format of expressions that solve handles; the symbol @ denotes list concatenation; and the auxiliary function `trunc_sqrt` denotes integer square root, rounded down[3].

Example 3.8: Continuing with Example 3.5, let us inspect the result of solving the expression $2 \times \mathsf{N} + \mathsf{row} \times \mathsf{N} + \mathsf{row} + 1$, assuming an upper limit of 43 to array mm. The constant 1 forces us to consider a memory block of size $43 - 1 = 42$. Thus, we invoke: solve_all 42 $[2 \times \mathsf{N}, \mathsf{row} \times \mathsf{N}, \mathsf{row}]$. Recursion gives us solve 14 $(2 \times \mathsf{N})$, solve 14 $(\mathsf{row} \times \mathsf{N})$, and solve 14 N.

[2]Following SML's syntax, we surround lists with brackets. Context shall make it clear when we refer to lists such as $[u_1x_1, \ldots, u_nx_n]$, or to integer ranges, e.g., $[l, u]$.

[3]For simplicity, we restrict the implementation of solve to square roots, but, in practice, when dealing with n products, e.g., $e_1 \times e_2 \ldots \times e_n$, we use n invocations of solve, each onto the n-th root of $e_k, 1 \leq k \leq n$.

```
1  datatype Exp = Var of string
2               | Add of Exp * int
3               | Cmu of Exp * int
4               | Mul of Exp * Exp
5               | Min of Exp * Exp
6               | Max of Exp * Exp
7
8  fun solve n (Var v)          = [(v, n)]
9   | solve n (Add (e, c))      = solve (n - c) e
10  | solve n (Cmu (e, c))      = solve (n div c) e
11  | solve n (Min (e0, e1))    = solve n e0 @ solve n e1
12  | solve n (Max (e0, e1))    = solve n e0 @ solve n e1
13  | solve n (Mul (e0, e1))    =
14    let
15      val nn = if n =< 1 then 0 else trunc_sqrt n
16    in
17      solve nn e0 @ solve nn e1
18    end
19
20 fun solve_all n Le = map (solve (n div length Le)) Le
```

Examples: v; $v + 2$; $v \times 2$; $v \times u$; $\min(v, u)$; $\max(v, u)$

Fig. 11. Finding minimum upper bounds to symbols.

The first call produces the list $[\mathsf{N}, 7]$, indicating that $\mathsf{N} \leq 7$. The second yields $[(\mathsf{row}, 3), (\mathsf{N}, 3)]$, indicating that $\mathsf{row} \leq 3$ and $\mathsf{N} \leq 3$. The third gives us $\mathsf{row} \leq 14$. Thus, the final solution is the list $[(\mathsf{N}, 7), (\mathsf{row}, 3), (\mathsf{N}, 3), (\mathsf{row}, 14)]$. This list lets us conclude that any pair of values (v_0, v_1), $v_0 \leq 3$ and $v_1 \leq 3$ satisfies the constraints imposed onto N and row.

In what follows, we present a number of theorems related to the correctness of the algorithm in Figure 11. Theorem 3.9 ensures termination of solve_all. The other theorems clarify particular aspects of the semantics of this function.

Theorem 3.9 (Termination): solve always terminates.

Proof: structural induction on the structure of the input Exp. The call of solve at line 7 of Figure 11 is not recursive; hence, it terminates. The other calls, at lines 8-11 are recursive; however, they always use strictly smaller inputs. □

Theorem 3.10 (Bounds): If e is an expression with k variables, and $L = [(v_1, n_1), \ldots, (v_k, n_k)] = \mathsf{solve}\ n\ e$, then $e \setminus L \leq n$.

Proof: the proof is by structural induction on e. There are five cases to consider:

- $e = \mathsf{Var}\ v$: in this case, $L = [(v, n)]$, and $e \setminus L = n$.
- $e = \mathsf{Add}(e', c)$: by induction, we assume that the hypothesis holds for $\mathsf{solve}\ (n-c)\ e'$; thus, $e' \setminus L \leq n-c$. But, $e = e' + c$; hence, $(e - c) \setminus L \leq n - c$, and so $e \setminus L \leq n$.
- $e = \mathsf{Mul}(e_0, e_1)$: if we let $n_q = \mathtt{trunc}\sqrt{n}$, then, by induction, we have that $n_0 = \mathsf{solve}\ n_q\ e_0 \leq n_q$ and $n_1 = \mathsf{solve}\ n_q\ e_1 \leq n_q$. Notice that $n_q < n$, because we only invoke solve recursively when $n > 1$. Thus, $n_0 \times n_1 < n$.
- $e = \mathsf{Min}(e_0, e_1)$ or $e = \mathsf{Max}(e_0, e_1)$. Induction gives us that $\mathsf{solve}\ n\ e_0 \leq n$ and that $\mathsf{solve}\ n\ e_1 \leq n$. Thus, both $\mathsf{Min}(e_0, e_1)$ and $\mathsf{Max}(e_0, e_1)$ are less than or equal n. □

Theorem 3.11 (Strictness): Let e be an expression with k variables, such that $L = [(v_1, n_1), \ldots, (v_k, n_k)] = \mathsf{solve}\ n\ e$. For any $n'_j < n_j$, we have that $(e \setminus L) \setminus [v_j \mapsto n'_j] \leq n$

Proof: the proof follows as corollary of Theorem 3.10. The data constructors used in the definition of the datatype Exp, e.g., Var, Add, Mul, Min and Max, are monotonic. Thus, smaller inputs yield smaller results. And Theorem 3.10 already ensures that $e \setminus L \leq n$ □

Corollary 3.12 (Generalization): If $L_e = [e_1, e_2, \ldots, e_k]$ is a list with k expressions, and $[L_1, L_2, \ldots, L_k] = \mathsf{solve\_all}\ n\ L_e$, then $e_1 \setminus L_1 + e_1 \setminus L_1 + \ldots + e_k \setminus L_k \leq n$

Finding Lower Bounds. Function solve_all in Figure 11 finds conservative upper bounds to the program symbols in a symbolic summation. We must also determine lower bounds for those variables. To this end, we use the following procedure: in the absence of any constraint, a variable is lower-bounded by zero. Otherwise, if a program symbol s is used in n sub-expressions such as $s + c_k, 1 \leq k \leq n$, then we set its lower bound to $\max(-c_1, \ldots, -c_n)$. Hence, we ensure that symbolic summations do not contain negative products.

Example 3.13: The expression $\mathsf{row} \times \mathsf{N} + (\mathsf{N} - 1)$ gives us the lower bound $\mathsf{N} \geq 1$. Notice that, if we had, for instance, the expression $\mathsf{N} \times (\mathsf{N} + 1)$, then we would have the lower bound $\mathsf{N} \geq 0$, and if we had the expression $(\mathsf{N}+1) \times (\mathsf{N}+2)$, then we would have $\mathsf{N} \geq -1$.

Can we solve any symbolic summation? The answer to this question is no: our ability to solve symbolic summations depends on the quality of the range analysis that we use. We can only solve symbolic summations in which every integer is bounded, i.e., they are not associated with the symbols $+\infty$ or $-\infty$. For instance, variable i in Fig. 1 has an unbounded upper limit. In our implementation, we have employed a *non-relational* range analysis. In this family of analysis, ranges are formed by invariant symbols only – they cannot include other variables. To solve the constraints in Figure 7, we use the algorithm proposed by Nazaré *et al* [26], which, in turn, reuses the symbolic operations proposed by Blume and Eigenmann [25]. This *semi-relational* implementation is the simplest known type of range analysis that fits our problem, and runs in time linearly proportional to the program size. Previous results have shown that this kind of analysis is able to bound approximately 75% of every integer used in SPEC CPU 2006, for instance [26]. More precise (and slower) implementations can take us further. For instance, Logozzo and Fähndrich have shown that their semi-relational "Pentagon" lattice is able to bound more than 88% of every memory access found in a comprehensive subset of the .NET framework [31]. Fully relational range analyses, such as "Octogons" [32] take this precision even further, although the extra gains have been reported to be small [31].

D. Generation of Stubs

A *stub* is a mock function, which generates inputs to a procedure that we want to test. We use SOLVE_ALL to generate stubs. Given a function f, with n memory accesses, we compute upper and lower bounds on these accesses using

the techniques discussed in Section III-C. Our automatically generated stub then works as follows:

1) Guess a size s_a for each array a used in f;
2) Collect every memory access performed upon a into a list of symbolic summations L_e;
3) Use solve_all s_a L_e to place upper bounds on the program symbols found in L_e;
4) Find lower bounds to the variables used in L_e.

Once bounds have been computed, the stub generates inputs for the target function. For each variable v, with lower bound l_c and upper bound u_c, the stub produces concrete values for v in the range $[l_c, u_c]$. The lower and upper bounds of v are determined by the initial guess of the array sizes that v indexes. Users can vary the size of these arrays gradually, always producing larger memory chunks, or can generate random values for the arrays. Example 3.14 shows a stub that follows this second modus operandi. The entire process, from symbolic range analysis to stub generation runs in time linearly proportional to the program size in practice. As evidence to scalability, Griffin-TG produces stubs for all the benchmarks that we use in Section IV in milliseconds.

```
1  int next_rand(int);
2  int bounded_random(int, int);
3  void solve_all(map<string, int>&, int, string);
4  map<string, int> bounds;
5  int _5EF0_get_vector = 0;
6  float* get_vector(int Width) {
7    return (float*)malloc(sizeof(float) * bounds["N"]);
8  }
9  float* convol(float* mm, int row, int col, int N) {
10   int i;
11   float* v = get_vector(N);
12   for (i = 0; i < N; i++)
13     v[i] = mm[(1+row)*(N+1)+i+1] * mm[1+col+(i+1)*(N+1)];
14   return v;
15 }
16 int main(int argc, char** argv) {
17   int num_tests = atoi(argv[1]);
18   for (int i = 0; i < num_tests; i++) {
19     _5EF0_get_vector = 1 + next_rand(_5EF0_get_vector);
20     solve_all(bounds, _5EF0_get_vector, "(2*N)+(row*N)+row+1");
21     solve_all(bounds, _5EF0_get_vector, "col+(N*N)+N+1");
22     int row = bounded_random(0, bounds["row"]);
23     int col = bounded_random(0, bounds["col"]);
24     int N = bounded_random(0, bounds["N"]);
25     float* mm = (float*)malloc(3 * _5EF0_get_vector);
26     convol(mm, row, col, N);
27   }
28 }
```

Fig. 12. Example of stub. Our testing framework is written in C++, but we analyze C syntax, e.g., the code in grey.

Example 3.14: Fig. 12 shows the code produced automatically to test the convol function seen in Fig. 1. This code has been edited for readability. The routines in lines 1-3 belong to our testing library. The STD map "bounds" holds the upper bounds of variables, which are computed based on the global variable _5EF0_get_vector (See the calls of solve_all at lines 20 and 21). The lower bounds of variables are invariant, and are written directly in the code – for instance, at line 19, when we adjust the size of the array adding 1 to it.

The Ranges of Pointers. The main consequence of Theorem 3.3 is that our range analysis correctly models the actual values that variables can assume during the execution of a program. From this observation we arrive at Corollary 3.16: if we only create arrays whose size encompasses the ranges of all the variables used to index it, then these indexing operations are safe. We say that two pointers, a_1 and a_2, belong into the same *family of pointer aliases* if they may alias according to the result of a given pointer analysis, such as Andersen's [33], or Steensgaard's [34]. If a memory access $a[E]$ or $*(a + E)$ occurs within the program, and $a \in A$, then the size of any element in A is at least as large as $R(E)$. Definition 3.15 formalizes this notion.

Definition 3.15 (Array Types): If every access to any member of a family of pointer aliases A is indexed by a set of expressions $\{E_1, \ldots, E_n\}$, then the type of any a in A is $typeof(a) = \Pi(B).T^B$, where, $B = R(E_1) \sqcup R(E_2) \sqcup \ldots R(E_n)$, and T is the type of the memory cells indexed by pointers in A.

Corollary 3.16: If the type of an array v is $\Pi(E).\tau^E$, and the size of v is greater than E, then every indexation of v happens within bounds.

Proof: The abstract interpretation framework implies that any valuation of the variables and symbols of the program belongs to the concretisation of the intervals that **rg** (rules in Figure 7) finds. The result follows from Definition 3.15, and the definition of the operator $\sqcup$, which implements union of symbolic ranges. □

IV. Case Studies

The ability to generate safe inputs for type-unsafe programming languages lets us improve performance evaluation benchmarks and tools. This section illustrates this potential with two case studies. In Section IV-A, we show how our techniques can be used towards providing cogent conclusions about the comparison of two different ways to run GPGPU programs. In Section IV-B, we show how we can improve the usage of Aprof [14], an input sensitive profiler. In these two case studies, we are dealing with different problems: in Section IV-A, the already well-known fact [35] that usual benchmark collections come with a small number of different testing inputs; in Section IV-B, the fact that most functions are invoked only a handful of times during the execution of a program, an observation that is challenging to empirical complexity analysis.

Griffin-TG. The results in this section have been obtained through Griffin-TG, a tool that implements the ideas discussed in this paper. This tool has been built on top of the parser used in the Qt Creator IDE. We chose this parser due to its ability to handle programs whose source code is partially available (the parser's original motivation is syntax highlighting). Thus, we can extract a routine from a C program P, and test this routine separately from P. This choice has greatly simplified the implementation of Griffin-TG; however, it also has limited its scope. There exists no alias analysis implemented in Qt Creator; and we did not want to pay its implementation cost.

Thus, Griffin-TG can only analyze arrays that are unambiguous, e.g., whose address has not been taken, or that are marked with the "restrict" keyword. Consequently, the routines that our implementation can –currently– analyze, tend to be small. Small, yet, non-trivial: as we show in Section IV-A, we handle every kernel in the PolyBench suite, for instance. We emphasize that the lack of an alias analysis is a limitation of our tool, not of our techniques. The only factors that limit our techniques are the quality of the symbolic range analysis (as discussed in Section III-D), and the structure of memory indexing expressions (as discussed in Section II). We have used Valgrind [36] to certify that no experiment reported in this section incurred into invalid memory accesses.

A. Performance Testing

Context: Annotation systems such as OpenMP [37], OpenACC [38], OpenHMPP [39] and OpenSs [40] let developers mark parts of C programs that can run in Graphics Processing Units (GPUs). To use the GPU, data must be transferred from the host CPU, and then back. Some annotations are used to specify which chunks of data must be copied. In 2013, Nvidia released the so called *CUDA Unified Memory*, which makes such explicit copies unnecessary. This capability consists in a runtime system that manages buffer allocation and data coherence between CPU and GPU memories. Following the taxonomy of Jablin *et al.* [41, Fig.1], CUDA Unified Memory is a fully automatic management system. It shields programmers from concerns about data movement when developing GPGPU applications.

Problem: which mechanism leads to the implementation of faster programs: manual annotations, or automatic memory management? A benchmark previously used to answer this question is PolyBench, a collection of 24 programs used in scientific computations, like image convolution and LU-decomposition. However, PolyBench provides one set of inputs. If we use only these inputs to answer our question, then we might obtain a misleading answer, as previous work has observed [42]. PolyBench was originally built in the early 2010's. Today, most of its available inputs lead to negligible execution times in modern GPUs. In this case, managed memory is typically faster, as there is not enough computation to amortize the cost of batched data transfers. Griffin-TG lets us get around this shortcoming.

Solution: We have used Griffin-TG to produce automatic inputs to all the 24 PolyBench kernels. One of these kernels had to be modified to be correctly executed: LU. The modification was necessary to remove an error of division by zero, which could happen in some executions: the contents of one of the input arrays was used as a divisor. Griffin-TG only provides guarantees on the relation between array size and indexing expressions – for the contents of data-structures, we use random values. Apart from this modification, Griffin-TG could produce correct data for all the 99 different arrays used in the 24 kernels. We have executed each kernel 2,000 times, always with different inputs. None of these executions incurred in any invalid memory accesses. Figure 13 shows these results for the Atax kernel.

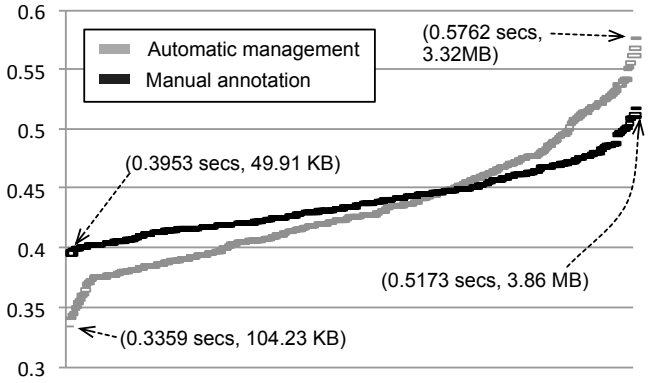

Fig. 13. Execution times produced for 2,000 executions of the ATAX kernel. Y-axis shows runtime, in seconds. X-axis shows execution samples, sorted by runtime. There is no correspondence between automatic and manual memory management on the X-axis. Largest observable input led to allocation of 3.86MB; smallest input 2.63KB. Each of the 4,000 samples was executed only once.

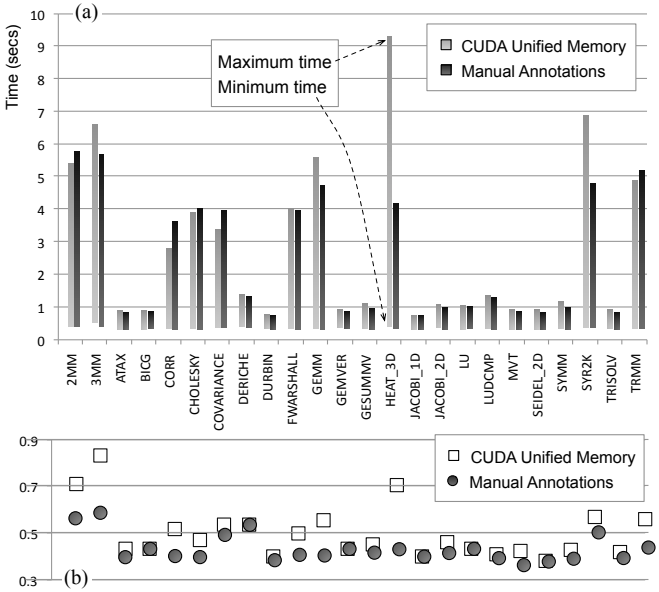

Fig. 14. (a) Maximum and minimum runtimes observed for all the PolyBench kernels. (b) Median value of runtimes observed for all the kernels (secs). We have removed outliers (2% lowest/highest samples).

Discussion: Given a kernel K, if we compare its two versions, managed and manually annotated, and we observe that the result of this comparison, for the largest inputs, matches the result for the lowest inputs, then we call this behavior *expected*, otherwise, we call it unexpected. In our experiments, we have observed expected behaviors in 15 of the PolyBench kernels. Figure 14 summarizes these findings. Thus, in nine of them, the result was unexpected; and, in eight of these situations the managed memory was faster for the smallest inputs, and slower for the largest – this last trend is also

observed in median and average values. The literature abounds with conclusions drawn from performance tests carried out on top of PolyBench. Our experiments show that the adoption of such conclusions demands care, and perhaps, even further examination.

B. Input-Sensitive Profiling

Context: *Input-Sensitive Profiling* is an expression coined by Coppa *et al.* [14], to denote a performance measurement methodology in which costs are associated with distinct input sizes. The key benefit of this methodology is the possibility of estimating, in a totally automatic and fully deterministic way, the asymptotic complexity of computational routines. The profiler receives a program P and its inputs, and produces a curve relating, for each function f in P, the memory that f reads, and the number of operations that it performs. The more often f is invoked, the more precise is the estimation of f's complexity.

Problem: to produce useful information, an input-sensitive profiler such as Aprof requires multiple executions of the same function with different inputs. If that is not the case, then it will not have enough data points to infer the routine's behavior. Unfortunately, during a typical execution of a program, the same function is invoked only a few times. Santos *et al.* [15, Fig.1] have showed that the number of invocations of the same procedure obeys a power distribution: over 50% of all the JavaScript functions in long browser sessions are called only once. Furthermore, 60% are called with the same arguments. Along similar lines, Coppa *et al.* [14, Fig.8] found out that less than 8% of all the functions in SPEC CPU2006's bzip2 are invoked more than 10 times.

Solution: Griffin-TG helps us to circumvent this shortcoming. To demonstrate this possibility, we adopt the following methodology: (i) we extract a function of interest from a program; (ii) we use Psyche-c [23] to convert this code snippet into a compilable program[4]; (iii) we use Griffin-TG to generate inputs for this program; and (iv) we use Aprof to estimate the asymptotic complexity of that code. To test this methodology, we have applied it to every one of the 24 kernels in the PolyBench suite: the function that denotes the kernel was tested apart from its enclosing program. Not even the code that initializes the kernel's input was present. Manual inspection reveals that in every case we have obtained the correct asymptotic complexity via Aprof.

The PolyBench kernels tend to be very homogeneous, from an algorithmic standpoint: they all range over vectors and (bi/tri-dimentional) matrices. Consequently, all the asymptotic complexity functions that we obtain are $O(n^k), k \in \{1, 2, 3, 4\}$. Thus, to exercise Griffin-TG in a larger context, we have also applied it onto sorting algorithms. Wikipedia[5] contains a list of 11 of them in a section called "Popular Sorting Algorithms". We have gotten implementations for all of them from a well-known open-source collection[6]. Thus, none of these implementations have been coded by ourselves. Using the combination of Griffin-TG+Psyche-c+Aprof, we could recover the complexity of all the implementations, except for "Radix Sort" and "Counting Sort". These algorithms contain memory accesses such as a[b[i]], which are indexed by an untraceable function input (see Definition 2.3). Figure 15 shows plots produced via Aprof for three of the sorting algorithms. These plots associate two quantities: the memory read by each algorithm (before being written, in case they suffer write operations), and the number of basic blocks executed by the algorithm[7].

Discussion: the methodology discussed in this section lets us analyze functions outside the programs that contain them. In addition to PolyBench and the sorting algorithms, we have applied it onto code snippets of larger programs such as FFmpeg, for instance. We claim that it would be difficult to reproduce these results even with state-of-the art test generators, such as KLEE [6], DART [7] or Austin [8]. Although such tools excel in code-coverage, they do not provide memory-access guarantees. To produce valid inputs, they must traverse the program from its beginning. Under such constraints, there is no guarantee that the function of interest will be invoked a number of times that is sufficient to enable the automatic inference of its complexity via input-sensitive profiling.

V. Related Work

This paper describes an example of Data-Flow Testing [43]. There exists a vast literature on the subject; however, most of this work concerns strongly typed languages, such as Java or JavaScript. In these languages, the size of an array is part of its definition, e.g., in Java, the size of array `a` is given by `a.length`. Thus, we do not need to resort to a static analysis to reconstruct this size as function of program symbols. Furthermore, out-of-bounds accesses are immediately detected once they happen; hence, not leading to undefined behavior.

To further distinguish our work from similar litetature, we point that several celebrated surveys on the subject of automatic testing do not discuss the problems posed by type-unsafe programming languages [44], [45], [46], [47]. This trend continues in textbooks, which often neglect undefined behavior in languages such as C, C++ and assemblies [48]. Edvardsson's survey briefly mentions that arrays and pointers complicate the generation of test cases [49], [50]. However, even without touching the problem of ensuring in-bound accesses, Edvardsson stated that this field was still open.

Nevertheless, if type-unsafe programming languages have not been the focus of those books and surveys, this omission does not mean that testing software implemented in these languages is not important. Testimony to the importance of this subject is the fact that tools such as KLEE [6], AUSTIN [8] and

[4]Psyche-c is a type inference engine for C. It reconstructs declarations that are missing in the code snippets that we have

[5]See Secs. 4.1-4.4 in `Sorting_algorithm` Rev. "21:32, 14 August 2017".

[6]Available at www.geeksforgeeks.org/sorting-algorithms/

[7]A basic-block is a maximal sequence of instructions $i_1, \ldots, i_n$, such that i_1 is the target of some jump, or the successor of a previous block, and i_n is a jump, or the predecessor of some block.

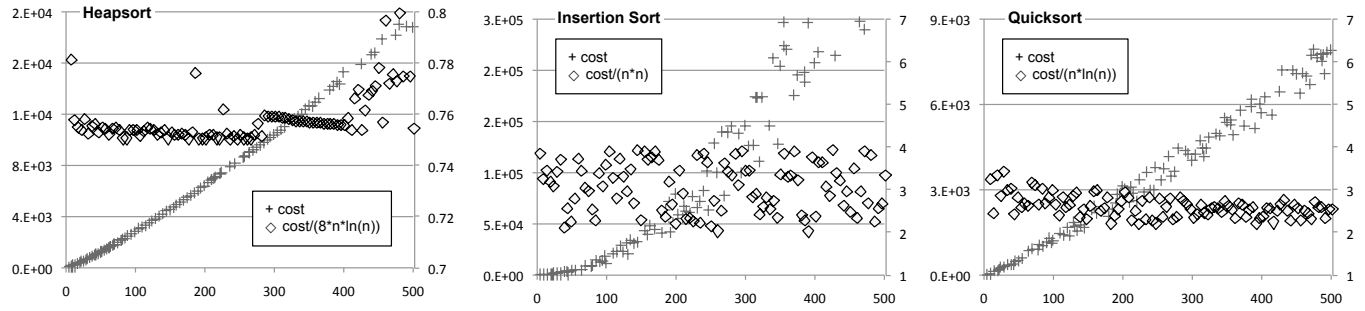

Fig. 15. Plots produced by Aprof for different sorting algorithms. The X-axis denotes *input size*: the memory read by the algorithm, in bytes. The Y-axis denotes *cost*: the number of basic blocks executed.

DART [7] either enjoy today much use, or have elicited much discussion. All these tools handle some type unsafe language, such as LLVM's intermediate representation [51] or standard C. However, although they excel in code coverage, they do not offer try to produce inputs that ensure in-bounds memory accesses – the goal of this work.

Load Testing. The testing methodology that we propose in this paper is especially useful in performance evaluation. In itself, this goal is already detached from the vast majority of the research in test automation, in which objectives are often related to bug finding à la *Sapienz* [52]. However, relatively recent work has brought renewed attention to performance bugs [53], [54], [55], [56]. In contrast to our work, these researchers' efforts focus on profiling techniques, rather than input generation. Therefore, we believe that our contribution can complement them. We notice that the software engineering literature contains work on test automation for performance evaluation [57], [58], [59]. Yet, the previous techniques that we know about have been applied onto Java programs: a setting in which the size of arrays is already part of their type by construction. Finally, there is much work on performance estimation [60], [61], [62], [63], [64]. For instance, Helal *et al.* propose a combination of static and dynamic analysis that predicts performance sequential code, once it runs in parallel [64]. An interesting future work is to use Griffin-TG to verify the accuracy of this kind of predictions.

Test of Code Partially Available. The ability to test partial code is important to the programming languages community. Researchers have already gone to great lengths to test code in face of missing parts [65], [66], [67]. The usual approach to compile an incomplete program relies on the creation of *stubs* [68], functionally minimal test-focused implementations of a software interface that replaces their real counterparts. However, the reconstruction of *memory-unsafe* languages, such as C, C++ or assembly poses challenges to the state-of-the-art techniques available today. In these languages, built-in data-structures lack size information, such as Java's array.length, C#'s array.Length, or Python's len(array). A notable work in unsafe languages is Godefroid's *Micro Execution* approach [66]: to test a program Godefroid runs it in a virtual machine, and whenever access to unallocated memory takes place, he allocates this memory on the fly. Micro Execution was designed to find bugs in low-level code. It is not possible to use it to control the size of memory blocks, in order to carry out performance tests, like we do. Additionally, we point out that our approach does not require any form of virtualization: it is completely static.

VI. Conclusion

This paper has presented a methodology to generate inputs for memory-unsafe languages that ensure inbounds accesses. Our ideas work for a wide range of actual programs, and complements previous techniques that seek to maximize coverage of test cases. In other words, we generate the size of arrays and other aggregate memory blocks, not their contents. Our ideas can be also used alone, without the support of other tools, as an effective way to carry out performance tests, as we have demonstrated empirically.

Our approach is consistent, but not complete: we handle (i) memory accesses indexed by expressions that can be transformed into monotonic functions, and that (ii) are bounded by the inputs that a function receives. Over 90% of the memory accesses that occur in actual programs obey (i), even under limited support of alias analysis. The quality of the range analysis that is coupled with out technique determines (ii). In this paper we used a non-relational implementation of symbolic range analysis, which already lets us deal, for instance, with every kernel in the PolyBench suite. How heavier algorithms impact into our approach is a question that we leave open as future work.

Software: Griffin-TG is available at https://github.com/maroar/griffin-TG.

Acknowledgment

This project was made possible by grants *16/2014 – Cooperação Multilateral FAPEMIG-INRIA-CNRS* and *CNPq-202896/2017-0 – Pós Doutorado no Exterior*. While working on this project, Marcus Rodrigues received a scholarship from CAPES. We thank Solène Mirliaz for working on an earlier version of Griffin-TG.

REFERENCES

[1] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *POPL*, (Washington, DC, USA), pp. 97–108, ACM, 2007.

[2] D. M. Ritchie, "The development of the c language," in *HOPL-II*, (New York, NY, USA), pp. 201–208, ACM, 1993.

[3] R. Bodík, R. Gupta, and V. Sarkar, "ABCD: Eliminating array bounds checks on demand," in *PLDI*, (New York, NY, USA), pp. 321–333, ACM, 2000.

[4] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *POPL*, (New York, NY, USA), pp. 128–139, ACM, 2002.

[5] J. Lakos, *Large-scale C++ Software Design*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996.

[6] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, (Berkeley, CA, USA), pp. 209–224, USENIX Association, 2008.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 213–223, ACM, 2005.

[8] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: An open source tool for search based software testing of C programs," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 112–125, 2013.

[9] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *ESOP*, (Berlin, Heidelberg), pp. 520–535, Springer-Verlag, 2007.

[10] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, pp. 20:20–20:27, Jan. 2012.

[11] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX*, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.

[12] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *CGO*, (Washington, DC, USA), pp. 213–223, IEEE, 2011.

[13] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, D. Frampton, L. J. Hendren, M. Hind, A. L. Hosking, R. E. Jones, T. Kalibera, N. Keynes, N. Nystrom, and A. Zeller, "The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 4, pp. 15:1–15:20, 2016.

[14] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI*, (New York, NY, USA), pp. 89–98, ACM, 2012.

[15] H. N. Santos, P. Alves, I. Costa, and F. M. Quintao Pereira, "Just-in-time value specialization," in *CGO*, (Washington, DC, USA), pp. 1–11, IEEE, 2013.

[16] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, (London, UK, UK), pp. 79–103, Springer-Verlag, 1996.

[17] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *PACT*, (Washington, DC, USA), pp. 106–115, IEEE, 1998.

[18] J. Robinson, "The undecidability of exponential diophantine equations," *Studies in Logic and the Foundations of Mathematics*, vol. 44, no. 8, pp. 12–13, 1966.

[19] A. Wiles, "Modular elliptic curves and fermat's last theorem," *Annals of Mathematics*, vol. 141, no. 3, pp. 443–551, 1995.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *TOPLAS*, vol. 9, no. 3, pp. 319–349, 1987.

[21] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *PLDI*, (New York, NY, USA), pp. 290–299, ACM, 2007.

[22] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *CGO*, (Washington, DC, USA), pp. 126–135, IEEE, 2009.

[23] L. T. C. Melo, R. G. Ribeiro, M. R. de Araújo, and F. M. Q. a. Pereira, "Inference of static semantics for incomplete C programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 29:1–29:28, 2018.

[24] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. Pereira, "Runtime pointer disambiguation," in *OOPSLA*, (New York, NY, USA), pp. 589–606, ACM, 2015.

[25] W. Blume and R. Eigenmann, "Symbolic range propagation," in *IPPS*, (Washington, DC, USA), pp. 357–363, IEEE, 1994.

[26] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira, "Validation of memory accesses through symbolic analyses," in *OOPSLA*, (New York, NY, USA), pp. 791–809, ACM, 2014.

[27] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *TOPLAS*, vol. 27, no. 2, pp. 185–235, 2005.

[28] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: Static and dynamic memory reference analysis," in *ICS*, (Washington, DC, USA), pp. 251–283, IEEE, 2002.

[29] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, (New York, NY, USA), pp. 238–252, ACM, 1977.

[30] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Berlin, Heidelberg: Springer-Verlag, 2005.

[31] F. Logozzo and M. Fähndrich, "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses," *Sci. Comput. Program.*, vol. 75, no. 9, pp. 796–807, 2010.

[32] A. Miné, "The octagon abstract domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, 2006.

[33] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[34] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL*, (New York, NY, USA), pp. 32–41, ACM, 1996.

[35] Q. Liu, X. Wu, L. Kittinger, M. Levy, and C. Jung, "Benchprime: Effective building of a hybrid benchmark suite," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 179:1–179:22, Sept. 2017.

[36] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, (New York, NY, USA), pp. 89–100, ACM, 2007.

[37] J. Jaeger, P. Carribault, and M. Pérache, "Fine-grain data management directory for openmp 4.0 and openacc," *Concurr. Comput. : Pract. Exper.*, vol. 27, no. 6, pp. 1528–1539, 2015.

[38] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First experiences with real-world applications," in *Euro-Par*, (Berlin, Heidelberg), pp. 859–870, Springer-Verlag, 2012.

[39] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. T. no, "Locality-aware automatic parallelization for GPGPU with OpenHMPP directives," *Inter. Journal of Parallel Programming*, vol. 44, no. 3, pp. 620–643, 2016.

[40] C. Meenderinck and B. Juurlink, "Nexus: Hardware support for task-based programming," in *DSD*, (Berlin, Heidelberg), pp. 442–445, Springer-Verlag, 2011.

[41] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *PLDI*, (New York, NY, USA), pp. 142–151, ACM, 2011.

[42] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. a. Pereira, "DawnCC: Automatic annotation for data parallelism and offloading," *Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 13:1–13:25, 2017.

[43] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 5:1–5:35, 2017.

[44] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[45] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[46] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[47] V. G. Yusifoğlu, Y. Amannejad, and A. B. Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology*, vol. 58, pp. 123–147, 2015.

[48] D. Graham and M. Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*. Boston, MA, US: Addison-Wesley Professional, 1st ed., 2012.

[49] J. Edvardsson, "A survey on automatic test data generation," in *Compse*, (Begijnhoflaan, Belgium), pp. 21–28, EAI, 1999.

[50] E. Kit and S. Finzi, *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM, 1995.

[51] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[52] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *ISSTA*, (New York, NY, USA), pp. 94–105, ACM, 2016.

[53] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, (New York, NY, USA), pp. 77–88, ACM, 2012.

[54] R. Mudduluru and M. K. Ramanathan, "Efficient flow profiling for detecting performance bugs," in *ISSTA*, (New York, NY, USA), pp. 413–424, ACM, 2016.

[55] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *ICSE*, (Piscataway, NJ, USA), pp. 562–571, IEEE Press, 2013.

[56] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *PLDI*, (New York, NY, USA), pp. 369–378, ACM, 2015.

[57] L. Fang, L. Dou, and G. Xu, "PerfBlower: Quickly detecting memory-related performance problems via amplification," in *ECOOP* (J. T. Boyland, ed.), vol. 37 of *LIPIcs*, (Dagstuhl, Germany), pp. 296–320, Schloss Dagstuhl, 2015.

[58] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *ICSE*, (Piscataway, NJ, USA), pp. 156–166, IEEE Press, 2012.

[59] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *ASE*, (Washington, DC, USA), pp. 43–52, IEEE, 2011.

[60] J. Brock, C. Ding, R. Lavaee, F. Liu, and L. Yuan, "Prediction and bounds on shared cache demand from memory access interleaving," in *ISMM*, (New York, NY, USA), pp. 96–108, ACM, 2018.

[61] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control," in *MICRO*, (Washington, DC, USA), pp. 485–496, IEEE, 2010.

[62] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *PACT*, (Washington, DC, USA), pp. 220–231, IEEE, 2003.

[63] C. Dubach, T. M. Jones, and M. F. P. O'Boyle, "Exploring and predicting the effects of microarchitectural parameters and compiler optimizations on performance and energy," *Trans. Embedded Comput. Syst.*, vol. 11, no. S1, p. 24, 2012.

[64] A. E. Helal, W. Feng, C. Jung, and Y. Y. Hanafy, "AutoMatch: An automated framework for relative performance estimation and workload distribution on heterogeneous HPC systems," in *IISWC*, (Washington, DC, USA), pp. 32–42, IEEE, 2017.

[65] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *OOPSLA*, (New York, NY, USA), pp. 313–328, ACM, 2008.

[66] P. Godefroid, "Micro execution," in *ICSE*, (New York, NY, USA), pp. 539–549, ACM, 2014.

[67] C. Yao, Y.-W. Wang, F. li, and Y.-Z. Gong, "A method of function modeling in accurate stub generation," in *ICSAI*, (Washington, DC, USA), pp. 1–8, IEEE, 2014.

[68] G. Meszaros, *xUnit test patterns: Refactoring test code*. London, United Kingdom: Pearson Education, 2007.