Program Lifting using Gray-Box Behavior

Bruce Collie
School of Informatics
University of Edinburgh
United Kingdom
bruce.collie@ed.ac.uk

Michael F.P. O'Boyle
School of Informatics
University of Edinburgh
United Kingdom
mob@inf.ed.ac.uk

***Abstract*—Porting specialized application components to new platforms is difficult. This is particularly true if the components depend on proprietary libraries, or specific hardware. To tackle this, existing work has sought to *recover* high-level descriptions of application components to ease their retargeting. However, existing schemes are either too limited, targeting just one application domain, or too weak, making them ill-suited to recovering real-world programs. Additionally, many rely on help in the form of problem-specific user annotations or complex specifications.**

This paper develops a new approach using *gray-box* program synthesis, which recovers code by automatically constructing a program to match the behavior of an unknown component. However, unlike other synthesis approaches, it exploits the dynamic or gray-box behavior of a component to guide recovery. For example, the execution time, memory access patterns or observed instruction traces can all be used to direct synthesis.

We evaluate our technique (HAZE) extensively against existing program synthesizers and a domain-specific lifter. Our scheme is able to generalize effectively across domains, synthesizing and lifting more programs than prior techniques, without any external assistance. We validate our methodology using bounded model checking, demonstrating that our synthesized programs are correct. Finally, we apply our approach to machine learning workloads, obtaining significant speedups automatically.

I. Introduction

We live in an age of rapid hardware evolution due to the decline of Moore's Law. More than ever, porting existing applications to new devices is critical if we wish to effectively exploit specialized hardware.

However, it is well known that porting application components to new hardware is difficult; components are often only available in a low-level form [4, 48]. It is even more challenging if they are proprietary libraries or implemented as specialized hardware [7].

This has led to a large body of work aimed at recovering high level code, to allow for retargeting to new systems. At the heart of these approaches is the idea of *lifting*, where high-level semantic information lost during the implementation of a component is reconstructed [14].

However, existing lifting schemes suffer from a number of problems which make them ill-suited to recovering real-world programs. Some are overly *specific*, limited to targeting just one application domain (e.g. image processing [37] or finite string automata [7]). Others, aiming at greater generality, are too *weak*, restricting the lifted programs to a few lines of loop-free code [5, 38] or Turing-incomplete DSLs [9]. Still others push the problem to the developer, either requiring *external* access to the binary [22]; a full specification and a oracle that can provide counterexamples [1, 10]; or relying on user-provided annotations to help the recovery [17].

What we want is a *general* lifting scheme able to recover computationally important programs. It should be *automatic*, requiring no external specifications or human assistance. Furthermore, given that the components to be recovered may be external libraries (or even implemented in hardware, we do not want to rely on static user-provided binaries.

This paper develops a new approach to lifting an unknown component using *gray-box* program synthesis. Unlike other synthesis approaches, which construct a program solely based on a static specification [40, 41], it exploits, wherever possible, dynamic observations of component behavior. In particular, it is able to use the execution time, memory traces and performance counters of a component to guide the synthesis of an equivalent program.

We applied our approach (HAZE) to real-world components. These included DSP kernels, image processing libraries and tensor computations. We extensively evaluated our technique against existing program synthesizers and a specialized lifter.

We show that our scheme is able to generalize effectively across domains, synthesizing and lifting more programs than prior techniques, without any external assistance. We demonstrate, using bounded model checking, that our programs are correctly synthesized. We applied our approach to machine learning workloads and showed that we can obtain speedups of up to $4\times$ automatically.

II. Overview

Our aim in this paper is to synthesize a program with behavior equivalent to an existing system component, for which little or no *external* specification is provided. We first give an illustrative example of our approach, which is then followed by a brief discussion of how it relates to standard approaches to synthesis.

A. Example

Consider a scenario where we have a call to an external library which we would like to replace with one to a newer, faster implementation. However, the library source code is not

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

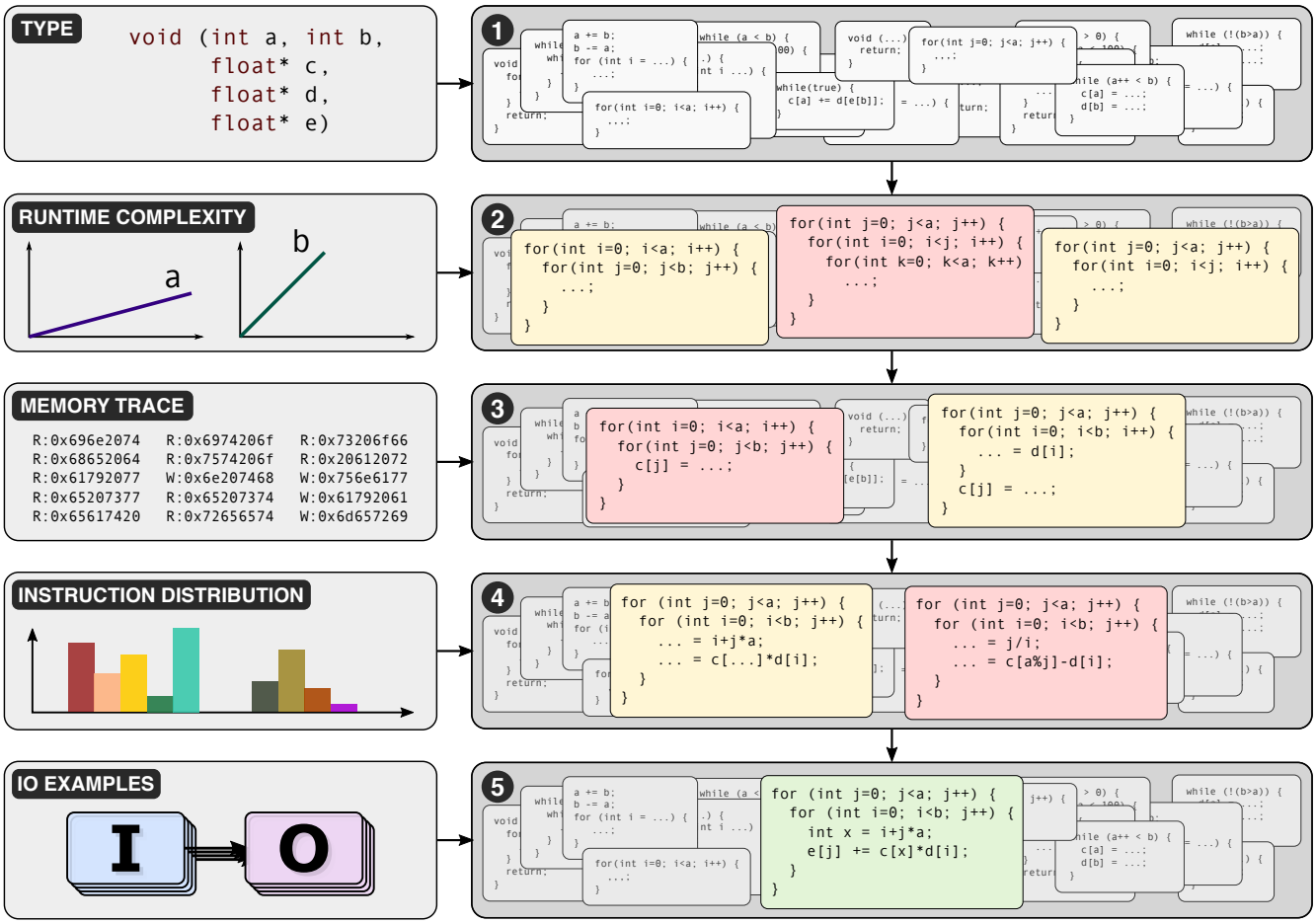


Fig. 1: The left column shows types of information used for disambiguating synthesis candidates, and the right hand column shows the space of potential candidates at each step. As more information is obtained, the space is narrowed, with candidates colored pink discarded due to their incompatibility with newly available gray-box information. Yellow candidates are compatible and pass onto the next stage. The green box shows the successful synthesized candidate that matches the input/output behavior of the component (dense matrix-vector multiplication).

available. We use *gray-box* information gathered from library execution to synthesize an equivalent program, which can be used to drive the desired refactoring. In this section, we make reference to Figure 1 throughout.

a) Type: We start with the target component type signature. It provides the structure of data passed into and out of the component, allowing the automatic construction of input examples. However, it does little to narrow the search space of potential implementations (stage ❶).

b) Runtime: When executing a component to check that its input-output (IO) functionality matches any candidate synthesized program, we can also easily observe its elapsed runtime. This gives insight into how the internal implementation uses the parameters it is passed: here, we observe that its runtime scales linearly in the value of both parameters `a` and `b` (i.e. it is both $\in \Theta(a)$ and $\in \Theta(b)$). This observation allows us to narrow the space of potential solutions significantly; the candidate in the pink box at stage ❷ can be discarded as it has complexity $\Theta(a^3)$, while the two yellow-box candidates remain.

c) Memory Traces: In many cases we can observe a trace of the memory addresses accessed during execution which can be used to further refine the candidate space. At stage ❸, for example, all traces contain a pattern of access to arrays `c` and `d`. This rules out almost all of the possible combinations of control flow identified previously: the code in the pink box can be discarded as it only refers to array `c`.

d) Performance Counters: Given the partial program structure above, the final step in our synthesis process is to work out the actual computation being performed by the component. Typically, synthesizers at this stage perform an expensive enumerative search.

If we can obtain *performance counters* from the component, we can produce a distribution of what types of operation or instruction it contains and direct the search. Here (at stage ❹), we observe that add and multiply instructions are approximately equal in frequency, and that other types (divisions, modulos, etc.) occur infrequently or not at all and can therefore

be discarded. With these observations, an enumerative search is able to quickly identify a correct solution (shown at stage ❺).

B. Synthesis for Lifting

Existing approaches to synthesis based lifting can be broadly characterized as white-box, black-box or multi-modal schemes.

a) White and Black-box Synthesis: In classical approaches, an external formal specification is provided which can be used to direct the construction of programs via, for example, counterexample-guided search (CEGIS) [1, 10]. We can characterize these as *white-box* schemes as extrinsic specifications are provided, and we can introspect the structure of candidate solutions for formal verification [16].

An alternative approach is *black-box* synthesis, or *programming by example* (PBE). Here, specifications are simply input-output examples that provide pointwise constraints on solution behavior. Formal verification is not possible, meaning *observational* equivalence is the strongest possible correctness guarantee. There exists a substantial, often machine learning-based body of work in this area [6, 25, 28, 29].

Unfortunately, from a systems perspective, both approaches are limited in the complexity of programs they can model, and are ill-suited to the complexity and specifications found in real-world problems, leading to the development of domain-specialized lifters [37]. However, real components provide other information which can guide gray-box synthesis.

b) Multi-Modal Specifications: Combining multiple specifications for a single synthesis problem has been explored in software engineering tasks such as API migration [42]. Similarly, the use of natural language descriptions alongside IO examples has recently been deployed in neural approaches to program synthesis [15, 38]. However, these approaches are *weak* (the programs are restricted to small DSLs), *specialized* (string processing and regular expressions respectively), and require *external* assistance in the form of human-provided text descriptions.

III. HAZE

Figure 2 shows an overview of our approach. All we assume is that the external component provides an API. From this, we generate inputs and call the component. It returns outputs, which are used for later observational equivalence testing. Varying inputs in a structured manner lets us observe how they affect execution time, which we model and use as a *sketch* [54] to guide synthesis. If available, memory traces and performance counters can be used to further refine the sketch and provide constraints on candidate data-flow.

Based on these constraints, we iteratively sample the space of possible programs, until we generate candidates that match the input/output (IO) behavior of the component. We can generate many such IO examples to ensure confidence in observational equivalence. In Section V, we discuss how to generate sensible inputs, and the use of model checking to verify that our programs are correctly synthesized.

A. Formalization

In this section, we briefly formalize the intuitive description above.

a) Program Spaces: Let $\mathcal{U}_\gamma$ be the set of all programs with a particular representation γ (for example, "loop-free C program" is a specific representation). Then, a space S_γ with representation γ is any subset $S_\gamma \subseteq \mathcal{U}_\gamma$. There are many possible spaces for each representation. Additionally, we equip each space with a probability distribution over its elements; this encodes the ordering of search during the synthesis process. Typically, this distribution is implicit when writing down a program space; we write S_γ^P to make explicit that a space has distribution P.

b) Transformations: A transformation $t : S_\gamma \to S_\phi$ is any function t that maps a space with representation γ to one with representation ϕ. Transformations perform *refinement* during the synthesis process by reducing the size of the relevant search space.

c) Priors: Transformations are almost always parameterized by relevant prior knowledge of some kind. We identify two key types of prior knowledge: *concrete* knowledge that acts as a filter on the target space, and *probabilistic* knowledge that induces a ranking distribution. We write:

$$t : S_\gamma \xrightarrow{(k,P)} S_\phi$$

for a transformation t parameterized by a concrete prior k and probabilistic prior P.

d) Sketches: A common category of representations in synthesis is that of *program sketches* [54]. Intuitively, sketches represent partial programs that must be instantiated with local details to produce a full solution program. In this paper, we treat sketches abstractly as n-ary tree-like structures, where each sketch can be composed with zero or more child sketches [18, 39, 56].

e) Synthesis: A single phase of synthesis can be written as:

$$S_\gamma \xrightarrow{(k,P)} S_\phi$$

Intuitively, this represents *refinement* of a search space by transforming to a different representation and integrating contextual accumulated knowledge. If we have suitable knowledge (k, P), instead of searching through a large space S_γ we can refine the search to a more tractable space S_ϕ.

IV. GRAY-BOX INFORMATION

This section describes the sources of dynamic gray-box information used by our program synthesizer: HAZE.

A. Type Signature

The first source of information used by HAZE is the API or type signature τ of the target function. It simply allows us to implement a simple filtering transformation (where as before, $\mathcal{U}$ is the universal set of all programs):

$$\mathcal{U} \xrightarrow{\tau} \{p \in \mathcal{U} \mid p \text{ has signature } \tau\}$$

The type signature allows safe interpretation of the inputs passed to or outputs received from the target component.

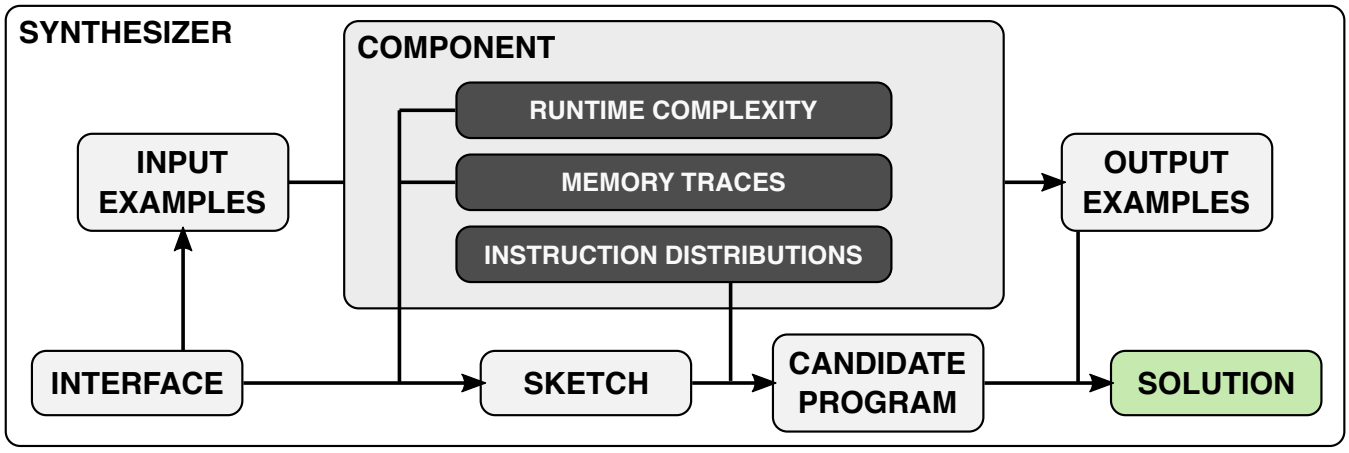


Fig. 2: Full system diagram showing how information flows through the stages of our synthesizer.

B. Generating Inputs

Schemes that rely on an *external* oracle to specify correctness often use randomly generated inputs over a restricted domain. While randomly generated inputs over a "small" restricted domain may exercise the required functionality for observational equivalence [19], they are unlikely to capture the full scope of dynamic, run time behavior.

1) Example: In Figure 3b, the "small" generation regime does not generate inputs large enough to confidently infer a performance model; for $n \in [-64, 64]$, the execution time is within measurement error and appears approximately constant.

One possible solution to this problem is to expand the sampling range; in Figure 3b, the "large" regime uses the range $[-1024, 1024]$ which is large enough to show clear increases in execution time. However, in this case the sampled values are clustered around $n = 0$, with only two exceeding $n = 300$. As a result of this nondeterminism, a large number of input values need to be sampled to consistently and confidently observe the quadratic complexity of `triangle_sum` (Figure 3a).

Our solution is to sample linearly spaced points from the same large input range, ensuring that we consistently capture the relevant system behavior. This method (the "linear" regime) is shown in Figure 3b.

C. Performance Models

There is a strong, intuitive correlation between a program's control flow structure and its runtime performance characteristics. To formally describe the runtime performance of our synthesis target, we use a notation similar to the *Performance Model Normal Form* due to Calotoiu et al. [12, 13]. This normal form provides a parameterized equation that can be used to describe the runtime performance of any program in terms of each of its scalar inputs $x_1, \ldots, x_m$ individually:

$$f(x_i) = c_0 \cdot c_1 x_i^{p_i} \cdot c_2 \log_2^{q_i}(x_i)$$

For each of these inputs, the runtime of the program at different values of the input is recorded. Then, exponents $p_i, q_i \in \mathbb{Q}$ and constant factors c_0, c_1, c_2 are regressed against the observed performance to produce the best-fitting performance model for each input parameter.

a) Sketches: We constructed a library of program sketches based on well-known algorithmic patterns, and associated each one with a runtime complexity class in terms of the parameters it uses. For example, two elements of the class $O(n^2)$ are:

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < n; ++j) { ... }
}

for(int i = 0; i < n; ++i) {
  for(int j = i; j < n; ++j) { ... }
}
```

Each sketch then provides a relationship between their own complexity and child complexity when combined. For example, nesting the two loop structures above gives a complexity of $O(n^2 \cdot n^2) = O(n^4)$.

b) Tree Search: A set of possible program sketches can be constructed using a simple tree search procedure. Initially, we partition the available partial sketches into two groups: those that have $O(1)$ complexity in terms of the function's parameters, and those that have greater than $O(1)$ complexity.

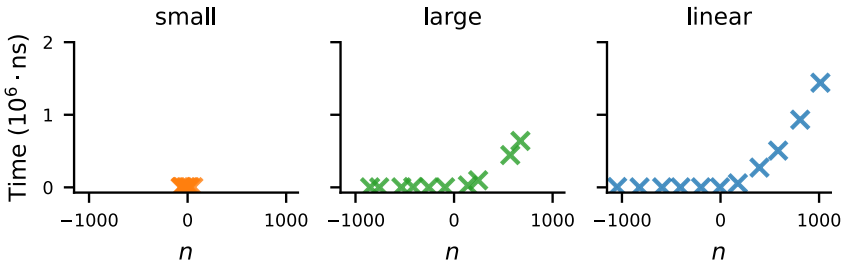
For each parameter's performance model, we identify a set of sketches that *could* contribute to that model. For example, a sketch with complexity $O(n^2)$ could appear if the overall complexity in n was $O(n^3)$, but not if it were only $O(n)$ (for example). Then, we enumerate compositions of the sketches for the full set of parameters, pruning compositions that exceed the total required complexity (see Figure 5 for an illustration of this).

Writing $\mathcal{S}$ for the space of all program sketches, the transformation we implement given an appropriate PCNF O is:

$$\mathcal{U} \xrightarrow{O} \mathcal{S}$$

```
int triangle_sum(int n) {
  int r = 0;
  for (int i = 1; i < n; ++i)
    for (int m = 1; m < i; ++m)
      r += m;
  return r;
}
```

(a) Simple example synthesis problem, with time complexity of $O(n^2)$ in its single parameter n [53].



(b) Execution times of `triangle_sum` plotted against 10 different input values of n, for three input generation strategies: small ($[-64, 64]$) and large ($[-1024, 1024]$) uniform random, and linearly spaced.

Fig. 3: Example showing the benefit of a partially-deterministic, interpretable input generation strategy when considering dynamic behavior of synthesis oracles.

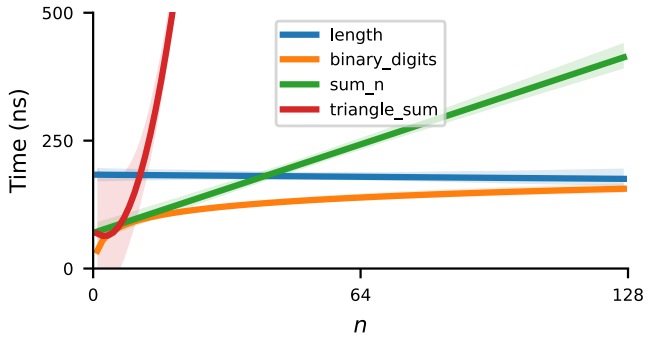


Fig. 4: Comparison of fitted runtime complexity models for four functions in our synthesis dataset. Using the linear sampling regime, complexity in a single parameter can be reliably inferred (shaded regions show 95% confidence intervals).

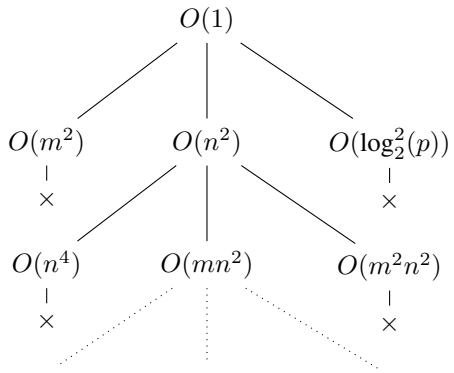


Fig. 5: Partial search tree for a sketch with overall complexity $O(mn^3\log_2(p))$ in parameters m, n, p. Each labeled node in the tree represents the accumulated complexity; paths through the tree are ruled out as their associated complexity becomes unviable.

D. Memory Traces

As well as execution time, for many systems, it is possible to observe an executing component's memory trace.

a) Traces: We define a memory trace $\overline{T}$ to be an ordered sequence of pairs:

$$\overline{T} \triangleq ((m_0, a_0), \ldots (m_n, a_n))$$

where each m_i indicates the type of access made (read or write), and each a_i indicates the address of that access. To minimize the burden of supplying this information, we do not encode the value read or written, nor the size of each access.

b) Normalization: The concrete memory addresses in a trace will be different even between executions of the same program, and so the traces need to be normalized to a common representation.

To do this, we first identify a set of base addresses in the trace, which all accesses will be expressed relative to. We do so by identifying addresses that appear at the beginning of contiguous sequences of accesses, using an assumption from related work in the area of memory trace analysis [49]. Then, we subtract the closest matching base address from each access to produce a (base, offset) pair, such that no offset is negative.

c) Sketch Tracing: We insert code into each sketch that imitates memory accesses at appropriate points. Then, we execute each sketch a number of times with this instrumentation, selecting randomly chosen assignments for branch conditions. This produces a set of memory traces for each sketch that represent a non-exhaustive possible set of that sketch's behaviors.

d) Alignment: Next, we use the Gotoh sequence alignment algorithm [27] to identify conserved regions between a sketch and program trace. This algorithm was selected specifically to deal with trace sections where long gaps appear; a likely scenario when considering traces generated from executing programs. The resulting alignment score represents how well the behaviors of the program trace are explained by the sketch trace. No alignment will score perfectly, but better correspondences will align more closely.

e) Scoring: From the target program we have a set $\mathbf{T}$ of ground-truth traces, and a set $\mathbf{T}_S$ of traces from each sketch S. From this, the aim is to select the sketch that explains the set of program traces best on aggregate. For an individual sketch trace $s \in \mathbf{T}_S$, we define a scoring function $q(s)$:

$$q(s) \triangleq \max\{\text{align}(s,t) \mid t \in \mathbf{T}\}$$

That is, the score for an individual sketch trace is simply its *best* alignment score with any program trace. This is intuitive; program traces may correspond to different executions, and so we cannot expect a single sketch trace to explain all possible program traces.

Then, for a sketch S with traces $s_1, \ldots, s_N$, we define an aggregated scoring function $Q(S)$:

$$Q(S) \triangleq \frac{1}{N}\sum_{i=1}^{N} q(s_i)$$

Sketches that score highest with respect to Q are those that on average, generate traces that best explain a trace from the set of program traces $\mathbf{T}$.

f) Ranking: We rank sketches from 1 to N according to their Q score, then assign each one a sampling probability using a geometric distribution:

$$\mathbb{P}(\text{sketch } k) \triangleq (1-p)^{k-1}p$$

HAZE uses $p = 0.5$, but p can be varied to reflect different priors on the structure of the ranked sketch set (i.e. if more sketches are likely to be viable, decrease p).

E. Instruction Type Distribution

Many synthesis procedures ultimately terminate in a resource-intensive enumerative search for a sequence of instructions or operations that comprise a correct solution. For HAZE, this entails searching for an instantiation of each hole in a program sketch with a concrete program value.

To accelerate this search for concrete instructions, HAZE considers the *observed* distribution of instruction types dispatched by the target component as it executes. By doing so, the synthesis process can be biased towards programs that produce similar distributions to the target.

More precisely, for a set of possible instruction opcodes:

$$O \triangleq \{o_0, o_1, \ldots, o_n\}$$

HAZE constructs a cumulative count C across all executions of the component:

$$C : O \mapsto \mathbb{N}$$

of how many times each opcode was observed during execution. From this count, HAZE then constructs a probability distribution $\mathbb{P}$ satisfying, for some $\delta \in [0, \frac{1}{n})$:

$$\forall i \,.\, \mathbb{P}(o_i) \sim C(i) \;\wedge\; \mathbb{P}(o_i) \geq \delta$$

```
int fact(int n) {
  int r = 1;
  while (n-- > 1) r *= n;
  return r;
}
```

Fig. 6: Synthesis problem demonstrating integer overflow; one possible cause of safety issues when generating inputs.

a) Search: The space of potential instantiations for a set of holes is combinatorially large. We sample a type-safe instruction opcode for each hole from the probability distribution described above, then enumerate possible arguments for that instruction using a set of heuristics (e.g. more local arguments are preferred to distant ones).

F. Gray-Box Synthesis

Our synthesizer, HAZE combines the transformations described above producing a complete pipeline that can be used to lift and synthesize programs. It uses LLVM intermediate representation as its target language (for compatibility with existing toolchains and boilerplate code [26]), which is then compiled and executed. Let $\mathcal{S}'$ be the space of fully instantiated instantiated sketches, and $\mathcal{L}$ be the space of LLVM programs. Then, using a distribution $\mathbf{P}$ and compilation function C we can write the transformation as:

$$\mathcal{S}' \xrightarrow{C} \mathcal{L}$$

We can now write down the full multi-phase synthesis pipeline implemented by HAZE:

$$\mathcal{U} \xrightarrow{\tau} \mathcal{U}_\tau \xrightarrow{O} \mathcal{S} \xrightarrow{(\mathbf{T},Q)} \mathcal{S}^G \xrightarrow{\mathbf{P}} \mathcal{S}' \xrightarrow{C} \mathcal{L}$$

V. SAFE SYNTHESIS

Given that we assume little of the target components, it is critical that the synthesis process is safe. In particular, we need to consider safe generation of component inputs and the correctness of synthesis.

1) Safety and Bounding: Not all programs can safely accept all possible values; some programs may exhibit incorrect or unsafe behavior when called with certain inputs. For example, the program listed in Figure 6 will overflow a standard 32-bit integer for values of n greater than 12. As well as integer overflow, there are safety issues for programs that access memory (for example, if a parameter is used as an index into an array also passed to the function). Where possible, our testing framework detects and reports these cases cleanly when collecting input-output examples; they are not included in the sets of IO examples used to specify correctness for a problem.

Fully random input generation methods struggle to concisely capture the behavior of reference functions in the presence of these safety issues. For example, the "small" and "large" fully random regimes in Figure 3b will both generate uninteresting ($n \leq 1$) or unsafe ($n > 12$) inputs, the majority

of the times they are invoked. This means that a large number of attempted inputs must be tried to produce the required number of working inputs.

Our solution uses the heuristic that programs exhibit such unsafe behavior for a continuous, infinite range of values either above or below a threshold. Under this assumption, when we observe a *run* of unsafe behavior, we back off and retry with the first observed unsafe value as our new upper bound on input. Intuitively, this compresses the same number of linearly spaced inputs into the safe interval supported by the oracle.

A. Verifying Synthesized Programs

In our *programming by example* setting, the only possible correctness specification for a synthesis problem is equivalent behavior over a set of input-output examples. This is known as *observational equivalence*; does the solution's behavior look the same for all the input examples attempted?

However, if the underlying code for a particular problem is available, we can use formal verification tools to provide stronger and more precise guarantees on the correctness of a solution. KLEE [11] is a suite of tools for performing *symbolic execution* on LLVM IR programs. By doing so, we are able to efficiently discover code paths within, or inputs to a program that cause errors to occur.

Our application of KLEE is to identify any inputs that cause two functions (i.e. a reference implementation and a candidate synthesized solution) to produce different behavior. We do so using a modified version of the basic function equivalence procedure suggested by Ramos and Engler [47], and with the key addition of a modernized implementation of the symbolic floating point support from KLEE-FLOAT [35].

VI. Experimental Setup

We constructed a dataset of problems from multiple sources and evaluated HAZE against two existing synthesizers and a lifter.

A. Dataset

a) Synthesis Problems: Our starting point is a set of 112 synthesis problems used to evaluate PRESYN [20]. The set of problems covers a range of domains (mathematical primitives, vector operations, string manipulations), and subsumes the evaluation sets for a number of other synthesizers [25, 38, 50, 53].

b) Benchmark Kernels: Suites of performance benchmarks are often used to evaluate binary lifting techniques [49]; we identified three such suites that provide a natural increase in complexity from the synthesis problems above. These were UTDSP [51], DSPStone [59] and PolyBench [45] ($N = 18, N = 15, N = 30$ respectively). The 63 new problems represent individual benchmark *kernels* that are more challenging than typical program synthesis problems, with the PolyBench set in particular containing some with far greater complexity than any existing synthesis techniques are able to scale to.

c) Specialized Domains: We identified a novel set of evaluation problems intended to model *real-world* code in two domains relevant to the synthesizers we evaluated HAZE against: image processing functions and tensor manipulations. We evaluated a set of $N = 9$ image-processing functions derived from [37] These covered both low-level implementation details of individual functions, as well as high-level heuristic descriptions (such as the "Blend" and "Filter" categories identified by Ahmad et al. [2]).

Understanding and manipulating tensor operations is an important task. We generated $N = 5$ common tensor manipulations using the Taco compiler [33, 34]. Each manipulation had a different set of optimizations and scheduling transformations applied to it in order to evaluate how well HAZE responds to changes in implementation detail.

B. Comparisons

We evaluate HAZE against the following leading program synthesizers and lifting approaches; PRESYN [20] exhibits state-of-the-art performance on the set of synthesis problems given above; SKETCHADAPT [38] is a leading example of a neural synthesizer, and *Helium* [37] is a domain-specific *lifter* for image-processing applications recovering high-level code from dynamic observations of an application executing.

C. Experiments

Because the synthesizers and lifters used in our evaluation target diverse problem domains and specification formats, running them all on the same set of benchmarks is challenging. This is a problem shared by other work in synthesis [20, 43].

a) Porting: All of the tools described above ran on our evaluation system without modification except Helium, for which we were unable to build successfully using its original toolchain. To resolve this, we produced a port of Helium's core that supported Linux binaries. This porting process was substantial in places, especially when dealing with platform-specific instrumentation tools.

b) Running Experiments: The core of our cross-evaluation is a comparison of which implementations were able to synthesize (or lift) each benchmark problem. To do so, we adapted each benchmark problem for the specific input requirements for each implementation. For example, PRESYN required training examples of previous syntheses, and Helium required inputs and outputs to be read from image-like files.

VII. Results

In this section we evaluate and analyze HAZE's synthesis performance against existing schemes. We examine how long HAZE takes to synthesize, and use model-checking to evaluate its validity. This is followed by an analysis of gray-box information and an ablation study on its use in synthesis. Finally, we examine a small case study that demonstrates the increased potential for performance improvement.

TABLE I: Summary of successfully synthesized or lifted programs across our evaluation dataset for each implementation examined. Columns show the number of successfully synthesized programs from each group for a single implementation.

	N	PRESYN	SKETCHADAPT	Helium	HAZE
Presyn	112	96	10	-	103
UTDSP	18	6	-	-	9
DSPStone	15	10	-	-	13
PolyBench	30	2	-	5	14
Image	9	2	-	10	6
Tensor	5	1	-	-	4
Total	189	117	10	15	149

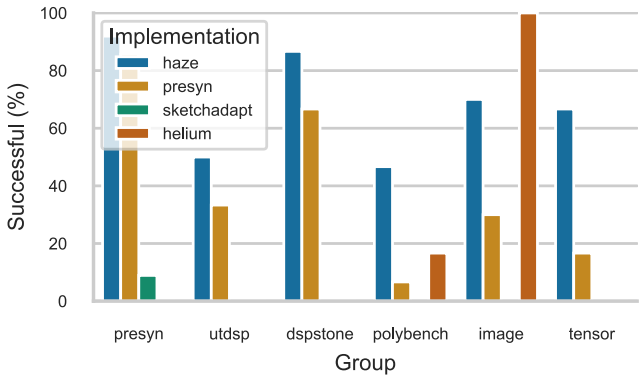


Fig. 7: Synthesis success rate for each synthesizer evaluated, across the set of benchmark suites used for evaluation.

A. Success Rate

For each implementation, we attempted each of the 189 synthesis problems listed in Section VI-A, recording the total successful syntheses for each implementation, as well as a per-group breakdown. These results are summarized visually in Figure 7, and listed in full in Table I.

HAZE is the best-performing implementation across the entire dataset, and on all but one of the individual problem groups (the image-processing kernels). It is also the only implementation able to synthesize at least one example from each of the benchmark groups.

By integrating multiple sources of gray-box knowledge, HAZE is able to outperform comparable implementations across the dataset at large. For example, HAZE is able to learn a group of logarithmic-time problems from PRESYN's benchmarks more effectively by using PMNF information (Section III). PRESYN fails to predict the control flow structure for these.

Helium outperforms all other implementations on its own specialized domain of image processing, but fails to generalize across the dataset. SKETCHADAPT successfully synthesizes a small number of simple examples,but is unable to scale in complexity or across domains. Given that it is targeted at list-processing tasks, this is not surprising.

Where HAZE fails, it does so most commonly because the sequence of instructions required to produce a correct solution

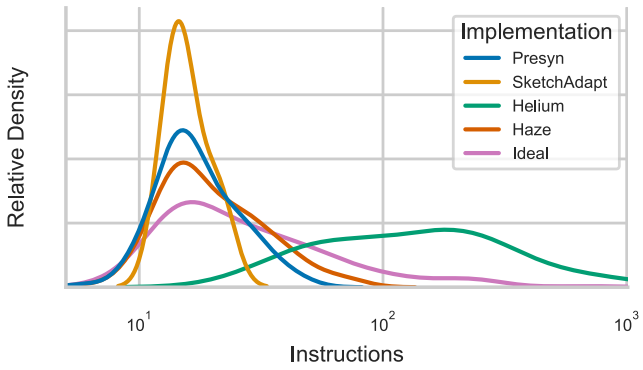


Fig. 8: Individually normalized kernel density estimate showing an approximate distribution for the number of instructions in successfully synthesized programs for each implementation, as well as instruction counts across the entire dataset ("Ideal"). The area under each curve is held constant, and so we can measure generalization across the dataset (i.e. adaptation to new problem domains) by similarity to the ideal curve.

to a problem is too long; even with the help of an probability distribution over their types, the space is too large to search effectively. Less commonly (and primarily in the PolyBench and UTDSP groups), the required control flow structures are not implementable using HAZE's library of sketches.

B. Analysis

Directly comparing the complexity of successful syntheses across implementations is challenging. To visualize the ability of an implementation to synthesize these complex examples relative to their frequency within the dataset, we computed normalized kernel density estimates for the number of instructions in each implementation's successful syntheses. These density estimates are shown in Figure 8, along with the density for the dataset as a whole. Qualitatively, the closer an implementation is to the dataset's curve, the less bias it exhibits towards programs of a particular size.

Existing synthesizers (PRESYN, SKETCHADAPT) show similar spikes at low instruction counts, indicating the difficulty they experience in scaling to larger problems. Conversely, Helium's distribution is centered at high instruction counts but does not cover small programs at all. An alternative view of the same information is given in Figure 9; it shows joint distributions of the number of instructions and the number of sketch fragments for the examples synthesized successfully by each implementation. Compared to existing implementations, HAZE takes a significant step towards generalizing across the entire dataset.

C. Synthesis Time

Optimizing for synthesis time was not a primary goal when implementing HAZE; for example, no directed search methods are used when generating instruction sequences. However, our results suggest that HAZE is usable. All our successful

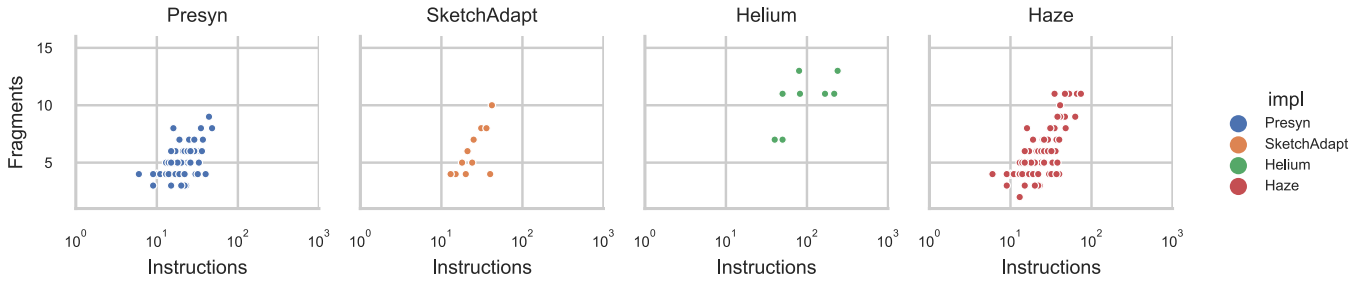


Fig. 9: Distribution of instruction count vs. fragment count for each implementation's successful syntheses; up and to the right represents more complex programs.

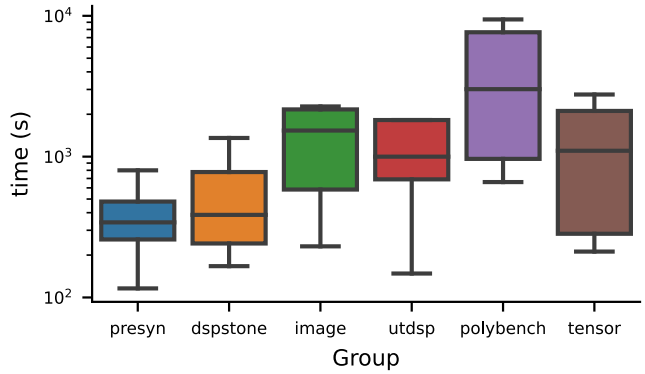


Fig. 10: Distributions of HAZE's required synthesis time across, each group of benchmark problems.

syntheses were obtained using a 3 hour threshold time on a desktop-class machine, and nearly 80% of these were obtained in under 15 minutes. Figure 10 shows the full distribution of time taken for successful syntheses by HAZE. It shows that the new datasets evaluated in this paper are significantly more complex, with PolyBench providing a particular challenge. Synthesis search time, rather than gathering IO examples and gray-box information, dominates the total time required to produce a solution (>99.9%). No problems required more than 1,000 IO examples to be generated.

D. Validity

In Section V-A, we described how the KLEE symbolic execution engine [11] can be used to model-check synthesized programs against a reference implementation (if one exists). Our results from the model-checking process are shown in Table II; we did not identify any programs synthesized by HAZE that exhibited *significantly* different behavior to the corresponding reference implementation.

More precisely, we found that many synthesized programs differed on minor floating-point arithmetic points when compared to the reference (for example, synthesizing $(a * b) * c$ instead of $a * (b * c)$ is technically incorrect in a non-associative arithmetic). Because our input-output example-based correctness checker adjusts for such cases using a ULP-based sliding equality check, such differences were common. These examples are summarized as **FP Assoc.** in Table II.

TABLE II: Results obtained from model-checking programs synthesized by HAZE against their respective reference implementations. Many solutions demonstrated minor floating-point inaccuracies that were explicitly not identified by observational equivalence checks. However, beyond these cases no false-positive synthesis results were identified.

	N	FP Assoc.	Bugs	Success
Presyn	103	11	0	100%
UTDSP	9	9	0	100%
DSPStone	13	13	0	100%
PolyBench	14	14	0	100%
Image	6	0	0	100%
Tensor	4	4	0	100%

After disregarding minor floating-point differences, no synthesized programs (i.e. those judged to be correct with respect to IO examples) were then judged to be incorrect by the model checker (**Bugs** in Table II). This validates our IO example-based approach to determining correctness.

E. Gray-Box Information

Figure 11 shows the complexity of components as synthesized by HAZE. These provide a strong signal in determining probable control-flow sketches. Similarly, Figure 12 shows how prevalent the 10 most common instruction types are in each of the 6 benchmark groups which guide data-flow selection when generating synthesized candidates. If we examine each group, we see that UTDSP has a similar distribution to Presyn with additional floating point arithmetic. Although DSPStone tackles the same DSP domain, its implementation produces a very different behavior with loads dominating.

F. Ablation Study

To determine the effect of each source of gray-box information on HAZE's synthesis performance, we performed a partial ablation study by omitting sources of information in turn.

We constructed four variants of HAZE: *Perf:* Runtime performance models only, no ranking of sketches, instructions are sampled uniformly; *Perf+Trace:* Runtime performance and memory traces; sketches are ranked but instructions

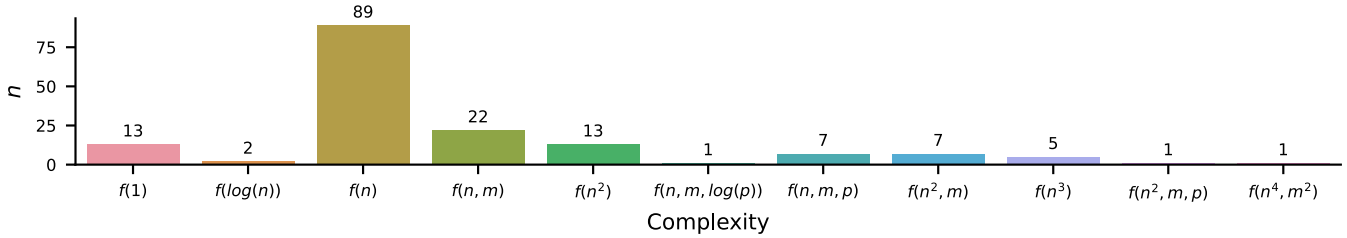


Fig. 11: Frequencies of observed potential complexity classes across our dataset of evaluation problems (with respect to scalar parameters). Names of parameters have been standardized to $n, m, p, \ldots$. The class $f(n, m)$ includes all classes that are both $O(n)$ and $O(m)$, for fixed m and n respectively (i.e. both $O(nm)$ and $O(n + m)$ are subsumed). Classes are ordered approximately from left to right by increasing complexity.

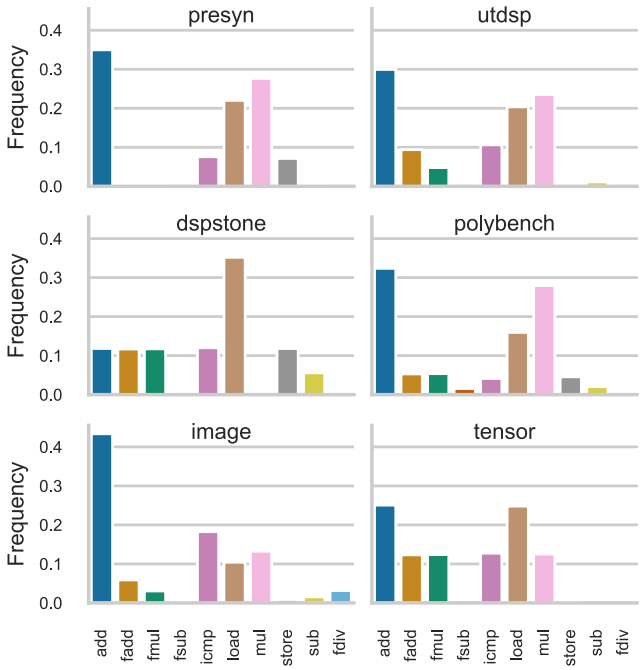


Fig. 12: Normalized distribution of the 10 most common instruction types for each group of benchmark problems.

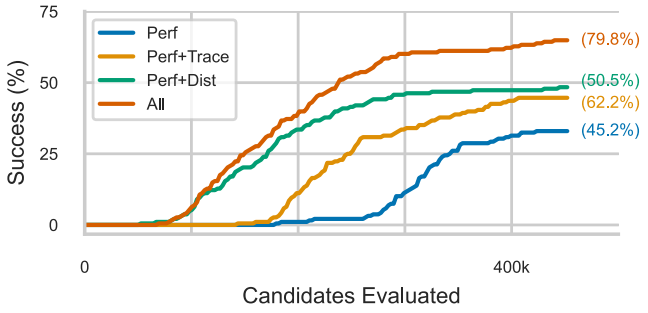


Fig. 13: Cumulative successful syntheses as a proportion of the entire evaluation dataset for four different versions of HAZE (baseline using only performance models to select sketches, adding memory traces and instruction distributions respectively, and the full HAZE pipeline). The distribution of successful syntheses is long-tailed, and so for clarity this figure shows only the initial phase where the majority of successes are achieved ($< 500,000$) candidates evaluated). Parenthesized values indicate the final success rate for each version; note that in extremis, the version using memory traces outperforms the one using instruction distributions.

are sampled uniformly. *Perf+Dist:* Runtime performance and instruction distributions; no sketch ranking but instructions are sampled from learned distributions. *All:* The full HAZE implementation.

For each variant of HAZE, we attempted to synthesize each problem in our evaluation dataset, and recorded the mean number of candidates required to do so successfully. These results are shown in Figure 13.

Adding each source of information produces a clear improvement to the achievable synthesis performance, though they do so through different mechanisms. *Perf+Dist* discovers programs with more instructions more quickly than the *Perf* baseline.Similarly, the *Perf+Trace* variant allowed programs with more complex control flow to be synthesized earlier.

Because problems with complex control flow are likely to also require more instructions to be correctly identified, the *Perf+Dist* variant requires more candidates to be evaluated before beginning to return successes. By combining the two approaches in HAZE, the initial phase of synthesis is as productive as *Perf+Dist* alone, but is able to scale to complex problems similarly to *Perf+Trace*.

G. Retargeting Case Study

Synthesized code can be searched for in an application [19], and refactored to use an improved compatible implementation. Although this paper is concerned only with the synthesis part of this workflow, we performed a small case study on tensor problems to demonstrate the benefit of our technique

a) Setup: We first evaluated the performance gained by replacing the Haze synthesized programs with observationally equivalent TACO GPU versions We then evaluated our ap-

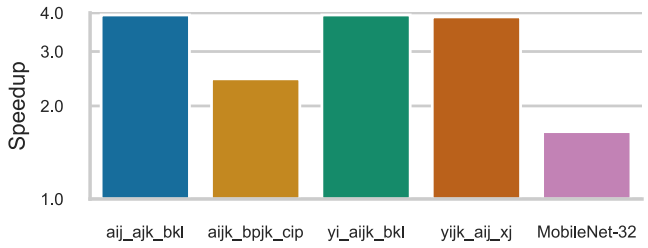


Fig. 14: Speedup achieved using a GPU implementation for each tensor operation synthesizable using HAZE.

proach on the MobileNet deep convolutional neural network, compiled to C for distribution to edge devices [57]. Here. we searched for instances of any of the successfully synthesized tensor operations and replaced it with a call to an optimized library implementation.

Our experiments were run on an Nvidia Jetson Nano developer kit using a Tegra T210 system-on-chip (4-core ARM CPU, integrated GPU, 8GB shared memory). The optimized tensor GPU implementations were generated using the Taco compiler [33], using a 256 GPU thread execution strategy.

b) Results: Each of the four tensor computations synthesizable by HAZE could be improved by moving execution from the CPU to the GPU on the test device, with speedups of 2–4×. as shown in Figure 14.

By replacing the MobileNet implementation of pointwise convolution with the TACO-generated library call, we were able to improve single-image inference time for the network by 60%. These results show that by improving synthesis ability, HAZE is able to enable new refactorings and increase performance.

VIII. Related Work

Prior work related to this paper can be broadly grouped as lifting and synthesis, although there is considerable overlap between the two areas.

A. Lifters

There are many papers targeting the lifting of low-level representations to a specific higher form; they can be characterized by the the type of low-level representation examined.

a) Code: Appropriately shaped kernels within existing legacy programs are often the target for lifting to a domain-specific language. For example, Kamil et al. [32] use a classical, syntax guided CEGIS procedure to detect loops compatible with Halide [46] in Fortran programs. Due to the complexity of establishing loop invariants, significant restrictions are made to reduce the search space. This work was later extended to C++ programs [2].

b) Binary: Other approaches have gone one step further, assuming that the source program is only available in binary form. Rather than a narrow, high-level DSL, some work tackles binary decompilation from binary to general purpose languages such as C [22]. In some cases, these techniques lean on ISA specifications to direct synthesis [30].

c) Memory-trace: An alternative approach makes fewer assumptions about the code, focusing instead on observable memory access behavior. Rodríguez et al. [48] analyze memory traces to recover polyhedral regions in code. This scheme is time-consuming, and fails when the accesses under analysis do not meet the strict requirements of the polyhedral model. In Helium, [37] a similar approach is used to find 2D stencil kernels in legacy image-processing binaries.

d) Other: Hardware behavior (such as power variation) has been used as a side-channel attack, allowing partial recovery of instruction sequences [23]. Such work requires extensive modeling of the system, and is aimed at recovering small instruction sequences for identification and attack rather than trying to synthesize large scale libraries.

B. Synthesis

a) Formal: Often, an external formal specification is provided to guide synthesis via, for example, CEGIS [1, 10]. Annotated type signatures or hints are often used to bias search towards correct programs, most commonly for functional programs [40, 41]. A key idea to making synthesis scalable is sketching [54, 55]. Sketching has been used in a large number of problem domains [24, 31, 52, 58] and allows the separation of high level program structure from detail. Typically, external hints or context are used to suggest structure, while enumerative search fills in the holes.

b) Machine Learning: Automatically generating programs from examples is a long-standing research area known as programming by example [28]. Recent work applying machine learning to this area has examined both induction (with a learned latent version of the program) and generation, which uses a language model to generate programs [3, 21, 44].

Rather than building a generative model, some use input-output examples to form priors over program distributions [8, 60]. However, schemes of this sort are typically limited to small DSLs and need large amounts of generated training data. Additionally, learned programs are often limited to a single problem domain for which a suitably-sized DSL exists (for example, list manipulation [38]).

c) Multi-modal: The use of multiple modalities of specification has been explored (for example, by using natural language text descriptions [36]). It has also been used for API migration, leveraging the large amount of work in natural language processing [42]. In SKETCHADAPT, such descriptions of programs are used alongside IO examples as multi-modal priors to drive probabilistic synthesis of string manipulation programs. More recently, Chen et al. [15] use IO examples and text descriptors in the domain of regular expression induction. However, in both cases, the approach does not generalize outside these restricted task-domains.

IX. Conclusion

This paper develops a new program lifting approach using *gray-box* behavior, automatically constructing a program to match the behavior of an unknown component. It generalizes across domains, synthesizing and lifting more programs than

prior techniques, without any external assistance. We validate our methodology using bounded model checking, demonstrating that our synthesized programs are correct. We apply our approach to machine learning workloads, obtaining significant speedups automatically. Future work will investigate other gray-box information to further improve performance.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[2] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.*, 38(6):204:1–204:13, November 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356549.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51 (4):81:1–81:37, July 2018. ISSN 0360-0300. doi: 10.1145/3212695.

[4] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. Binary analysis overview. In *Binary Code Fingerprinting for Cybersecurity*, pages 7–44. Springer, 2020.

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[6] Shengwei An, Sasa Misailovic, Roopsha Samanta, and Rishabh Singh. Augmented Example-Based Synthesis, 2019.

[7] Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. Accelerating Legacy String Kernels via Bounded Automata Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 235–249, Lausanne, Switzerland, March 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378503.

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. November 2016.

[9] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[10] Eric Butler, Emina Torlak, and Zoran Popović. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 10:1–10:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5319-9. doi: 10.1145/3102071.3102084.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[12] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 1–12, New York, NY, USA, November 2013. Association for Computing Machinery. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503277.

[13] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. Fast Multi-parameter Performance Modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181, September 2016. doi: 10.1109/CLUSTER.2016.57.

[14] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE.

[15] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 487–502, 2020.

[16] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[17] B. Collie, P. Ginsbach, and M. F. P. O'Boyle. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 55–67, September 2019. doi: 10.1109/PACT.2019.00013.

[18] Bruce Collie and Michael O'Boyle. Retrofitting Symbolic Holes to LLVM IR. *arXiv:2006.05875 [cs]*, June 2020.

[19] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F. P. O'Boyle. M3: Semantic API Migrations. In *Proceedings of the Thirty-Fifth International Conference on Automated Software Engineering*, ASE '20, Virtual Event, Australia, 2020. ACM. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416618.

[20] Bruce Collie, Jackson Woodruff, and Michael F. P. O'Boyle. Modeling Black-Box Components with Probabilistic Synthesis. *arXiv:2010.04811 [cs]*, October 2020. doi: 10.1145/3425898.3426952.

[21] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing Benchmarks for Predictive Modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 86–99, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8.

[22] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–671, 2020.

[23] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a Side Channel Based Disassembler. In Marina L. Gavrilova, C. J. Kenneth Tan, and Edward David Moreno, editors, *Transactions on Computational Science X: Special Issue on Security in Computing, Part I*, Lecture Notes in Computer Science, pages 78–99. Springer, Berlin, Heidelberg, 2010. ISBN 978-3-642-17499-5. doi: 10.1007/978-3-642-17499-5_4.

[24] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 572–585, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062382.

[25] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737977.

[26] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 151–162, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179515.

[27] Osamu Gotoh. Optimal Sequence Alignment Allowing for Long Gaps. *Bulletin of mathematical biology*, 52(3): 359–373, 1990.

[28] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL http://doi.acm.org/10.1145/1926385.1926423.

[29] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press, January 2016. URL https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/.

[30] Niranjan Hasabnis and R Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–324, 2016.

[31] Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. Synthesis of Recursive ADT Transformations from Reusable Templates. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 247–263. Springer Berlin Heidelberg, 2017. ISBN 978-3-662-54577-5.

[32] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908117.

[33] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 943–948, Urbana-Champaign, IL, USA, October 2017. IEEE Press. ISBN 978-1-5386-2684-9.

[34] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1–77:29, October 2017. doi: 10.1145/3133901.

[35] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 601–612, Urbana-Champaign, IL, USA, October 2017. IEEE Press. ISBN 978-1-5386-2684-9.

[36] Mehdi Manshadi, Daniel Gildea, and James Allen. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pages 661–667, 2013.

[37] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737974.

[38] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to Infer Program Sketches. In *International Conference on Machine Learning*, pages 4861–4870, Long Beach, CA, USA, May 2019.

[39] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed

holes. *Proceedings of the ACM on Programming Languages*, 3(POPL):14:1–14:32, January 2019. doi: 10.1145/3290327.

[40] Peter-Michael Osera. Constraint-based Type-directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2019, pages 64–76, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6815-5. doi: 10.1145/3331554.3342608.

[41] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738007.

[42] Rahul Pandita, Raoul Praful Jetley, Sithu D Sudarsan, and Laurie Williams. Discovering likely mappings between apis using text mining. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 231–240. IEEE, 2015.

[43] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1589–1596, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4939-0. doi: 10.1145/3067695.3082533.

[44] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. November 2016.

[45] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437, 2012.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176.

[47] David A Ramos and Dawson R. Engler. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5.

[48] Gabriel Rodríguez, José M Andión, Mahmut T Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 139–149, 2016.

[49] Gabriel Rodríguez, Mahmut T. Kandemir, and Juan Touriño. Affine Modeling of Program Traces. *IEEE Transactions on Computers*, 68(2):294–300, February 2019. ISSN 2326-3814. doi: 10.1109/TC.2018.2853747.

[50] Christopher D. Rosin. Stepping Stones to Inductive Synthesis of Low-Level Looping Programs. *arXiv:1811.10665 [cs]*, November 2018.

[51] R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications*. PhD thesis, 1998.

[52] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular Synthesis of Sketches Using Models. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 395–414. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54013-4.

[53] Sunbeom So and Hakjoo Oh. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In Francesco Ranzato, editor, *Static Analysis*, Lecture Notes in Computer Science, pages 364–381, Freiburg, Germany, 2017. Springer International Publishing. ISBN 978-3-319-66706-5.

[54] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 4–13. Springer, Berlin, Heidelberg, December 2009. doi: 10.1007/978-3-642-10672-9_3.

[55] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, October 2013. ISSN 1433-2787. doi: 10.1007/s10009-012-0249-7.

[56] Emina Torlak and Rastislav Bodik. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509586.

[57] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey. Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 101–110, September 2018. doi: 10.1109/IISWC.2018.8573503.

[58] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062365.

[59] V. Zivojnovic, J. Martinez, C. Schläger, and Heinrich Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proc. of ICSPAT'94 - Dallas*, October 1994.

[60] Amit Zohar and Lior Wolf. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages

2098–2107, USA, 2018. Curran Associates Inc.