Scaling up Superoptimization

Phitchaya Mangpo
University of California, Berkeley
mangpo@eecs.berkeley.edu

Aditya Thakur
Google Inc.
avt@google.com

Rastislav Bodik
University of Washington
bodik@cs.washington.edu

Dinakar Dhurjati
Qualcomm Research
dinakard@ti.qualcomm.com

Abstract
Developing a code optimizer is challenging, especially for new, idiosyncratic ISAs. Superoptimization can, in principle, discover machine-specific optimizations automatically by searching the space of all instruction sequences. If we can increase the size of code fragments a superoptimizer can optimize, we will be able to discover more optimizations.

We develop LENS, a search algorithm that increases the size of code a superoptimizer can synthesize by rapidly pruning away invalid candidate programs. Pruning is achieved by selectively refining the abstraction under which candidates are considered equivalent, only in the promising part of the candidate space. LENS also uses a bidirectional search strategy to prune the candidate space from both forward and backward directions. These pruning strategies allow LENS to solve twice as many benchmarks as existing enumerative search algorithms, while LENS is about 11-times faster.

Additionally, we increase the effective size of the superoptimized fragments by relaxing the correctness condition using contexts (surrounding code). Finally, we combine LENS with complementary search techniques into a cooperative superoptimizer, which exploits the stochastic search to make random jumps in a large candidate space, and a symbolic (SAT-solver-based) search to synthesize arbitrary constants. While existing superoptimizers consistently solve 9–16 out of 32 benchmarks; the cooperative superoptimizer solves 29 benchmarks. It can synthesize code fragments that are up to 82% faster than code generated by gcc -O3 from WiBench and MiBench.

Categories and Subject Descriptors D.1.2 [Automatic Programming]: Program Transformation; D.3.4 [Programming Languages]: Processors-Optimization

Keywords Superoptimization, Program Synthesis, SMT

1. Introduction

Code optimization is more important today than ever before. For example, CERN’s internal study demonstrated that using a highly optimizing compiler with profile-guided optimizations increased the power efficiency of its data center by 65% [15]. Another study shows that loop optimizations alone improved energy consumption of applications running on battery-operated portable devices by up to 10 times [16]. Code optimizers may also reduce costs of devices by enabling developers to select lower-power computing resources and smaller memory [7].

Developing a code optimizer still remains a challenging problem. The task of implementing a code optimizer is further exacerbated by the development of different instruction set architectures (ISAs) for different types of processors. For example, ARM alone has over 30 variants of ISAs [35], and new architectures are constantly being developed [8, 10, 12, 19, 22, 29, 36]. Even when compiling for widely-used architectures, like x86 or ARM, compilers may miss some optimizations that human experts can recognize. Many of these optimizations are local and very specific to the architectures. Although the expert developers can specify peephole optimizations in the compilers to perform these local rewrites, they may still miss some optimizations, and their rewrite rules may be buggy [17].

Superoptimization, introduced by Massalin [18], is a program optimization technique that searches for a correct and optimal program given an optimality criterion, instead of relying on rewrite rules. Thus, a superoptimizer can be used for automatically generating peephole optimization rules for compilers [5, 11] or optimizing small sequences of instructions produced by compilers on the fly [1, 24, 25]. With this technique, we can avoid buggy human-written rewrite rules and potentially discover even more optimizations. Note that superoptimization subsumes instruction selection, instruction scheduling, and local register allocation. A superoptimizer is shown to optimize a complex multiplication kernel and offer 60% speedup over an optimizing compiler [24].

Our aim is to develop a search technique that can synthesize optimal programs more consistently and faster than existing techniques. We experimented with the most common superoptimization search techniques: symbolic (SAT-solver-
based) [27, 31], enumerative [5, 6, 9, 11, 18, 32, 34], and stochastic [24, 25] search. A symbolic search could synthesize arbitrary constants, but it was the slowest. An enumerative search synthesized relatively small programs the fastest, but it could only synthesize up to three ARM instructions within an hour. A sliding window decomposition [20] could scale symbolic and enumerative search to larger programs, but it does not guarantee the optimality of the final output programs. A stochastic search could synthesize larger programs compared to symbolic and enumerative search, but it sometimes could not find the optimal program. This is because a stochastic search can get stuck at local minima.

We develop LENS, an enumerative search algorithm that rapidly prunes away invalid candidate programs. It employs a bidirectional search to prune the search space from both forward and backward directions. It also refines the abstraction under which candidates are considered equivalent selectively via an incremental use of test cases. In our experiment, these pruning techniques increase the number of benchmarks the enumerative search can solve from 11 to 20 (out of 22) and offer 11x reduction on the search time on average.

Although LENS performs better than the existing enumerative algorithms, it still cannot synthesize ARM code with more than five instructions or GreenArrays (GA) [12] code with more than 12 instructions. To scale this search algorithm to synthesize larger code, we introduce a context-aware window decomposition. With this decomposition, our enumerative search can synthesize an optimal (or nearly optimal) ARM program of 16 instructions within 10 minutes.

Optimizing code may require creating new constants or transforming the code fragment globally, which cannot be achieved by the enumerative search with the window decomposition. Thus, we compensate these limitations by combining stochastic and symbolic search into our superoptimizer, yielding a cooperative superoptimizer.

Finally, we develop GREENTHUMB, a framework for constructing superoptimizers for different architectures and testing different search techniques. The framework nicely facilitates testing various search techniques against each other and ensuring they work well for various ISAs. We instantiate GREENTHUMB for two very different ISAs—ARM and GA—for the testing purpose.

This paper makes the following contributions:

- the LENS algorithm, a bidirectional enumerative search with selective refinement (Section 3)
- the context-aware window decomposition, which scales a superoptimization technique that can optimize relatively small programs to larger programs (Section 4)
- the cooperative superoptimizer, which exploits strengths of different search techniques (Section 5)
- GREENTHUMB, a framework for constructing superoptimizers that provides efficient back-end search algorithms and can be extended to new ISAs (Section 6)

2. Overview and Insights

Figure 1 displays the interaction between the LENS algorithm (Section 2.1), the context-aware window decomposition (Section 2.2), and the cooperation of multiple search instances (Section 2.3), which can either employ LENS or different search techniques. The terminology used in this paper is defined as follows.

- A program is a sequence of instructions without loops and branches.
- A reference program is a program to be optimized.
- A program state contains values in the locations of interest such as registers, stacks, and memory.
- A test input is a program state that is used for checking correctness (being equivalent to a reference program).
- A test output is an expected program state after executing a candidate program on a given test input. A pair of a test input and a test input constitutes a test case.
- An equivalence verification is a process to verify if a candidate program is equivalent to a reference program on all inputs using a constraint solver.

2.1 Search Technique

A search technique searches for a program that is semantically equivalent to a reference program but faster according to a given performance model. This section provides our insights on how we design our search technique.

2.1.1 Problem Formulation

Let \( p_{\text{spec}} \) be a program we want to optimize. The set of test inputs \( I = (i_1, \ldots, i_n) \) and test outputs \( O = (o_1, \ldots, o_n) \) can be generated. Each test case \( (i_k, o_k) \) is an input-output pair such that \( p_{\text{spec}}(i_k) = o_k \). We formalize the superoptimization problem as a graph search problem. A node \( u \) in the graph represents a vector of \( n \) program states. The initial node \( s \) represents \( I \), and the goal node \( t \) represents \( O \). There is an edge from node \( u \)—representing program states \( (x_1, \ldots, x_n) \)—to node \( v \)—representing program states \( (y_1, \ldots, y_n) \)—labeled with an instruction \( \text{inst} \).
if \( \text{inst}(u) = v \), an abbreviation for \( \bigwedge_{i=1}^{n} \text{inst}(x_i) = y_i \). We use \( u \leadsto v \) to denote a set of all paths from \( u \) to \( v \), which represents a set of instruction sequences. A program that passes all \( n \) test cases corresponds to a path from \( s \) to \( t \). Therefore, the superoptimization problem reduces to searching for a path \( p \) from \( s \) to \( t \) such that \( \text{cost}(p) < \text{cost}(p_{\text{spec}}) \). We use \( q \otimes r \) to denote concatenation of programs \( q \) and \( r \).

2.1.2 Enumerative Search Algorithms

In this section, we illustrate the differences between existing enumerative algorithms and the LENS algorithm. Assume we want to synthesize an ARM program of four instructions using only two registers. A program state is represented by \( \langle r0, r1 \rangle \). Figure 2 shows the search graphs constructed by different algorithms, which will be explained in detail.

**Existing Algorithms** Enumerative algorithms enumerate all possible programs whose cost is less than \( \text{cost}(p_{\text{spec}}) \) and search for a program that is equivalent to \( p_{\text{spec}} \). The existing successful enumerative program synthesizers [2, 3, 6, 9, 32] apply an equivalence class concept, grouping programs into equivalence classes based on their behaviors on a set of test inputs. The search enumerates all possible behaviors, which can be many orders of magnitude fewer than all possible programs. Grouping programs based on a set of test cases is effectively abstracting the search space. The fewer the test cases, the more abstract the equivalence classes are; each equivalence class may contain more programs that are, in fact, not equivalent. Node \( u \) in the search graph essentially corresponds to the equivalence class containing programs \( s \leadsto u \), which have the same behavior according to the set of inputs \( I \).

The SIMD synthesizer [6] and the SyGus enumerative solver [3] are enumerative synthesizers that solve similar problems to ours. Both synthesizers use equivalence classes in a similar way to prune the search space. Here, we will explain their pruning strategy using our new formulation. Let \( p \) be a program prefix from \( s \) to \( u \). The algorithm searches for a program postfix \( q \) such that \( q(u) = t \). If there is no such \( q \), the search can prune all program prefixes in the same equivalence class as \( p \) away. The top part of Figure 2(a)
illustrates this idea. \( s_1 \leadsto c_1 \) corresponds to programs in the same equivalence class. The algorithm only needs to explore the subgraph rooted at \( c_1 \) once to prune away all paths from \( s_1 \) to \( c_1 \).

We observed two main sources of inefficiency in the existing algorithms. The first source of inefficiency comes from restarts. A restart happens when the search finds a feasible program, a program that passes the current set of test cases but is not equivalent to \( p_{spec} \); the abstraction is too coarse. The counterexample generated by a constraint solver is added to the test cases to refine the abstraction, and the search restarts building a new graph from scratch with respect to the updated \( I \) and \( O \). Upon restarting, the search forgets which programs it has already pruned away, so it revisits them again. Figure 2(a) illustrates that the search revisits programs from \( s_1 \) to \( c_1 \) in the new graph. Conceptually, when a new counterexample is found, the algorithm redivides the search space entirely as shown in Figure 3(a). The figure visualizes the space of all programs of size \( d \) (four in the example in Figure 2) divided into equivalence classes.

The second source of inefficiency comes from using more test cases than necessary. Consider programs \( p_1 \) and \( p_2 \) whose behaviors are the same on the first test case but different on the second one. If there is no \( q \) such that \((p_1 \oplus q)(I[1]) = O[1]\) with respect to the first test case, the search can also prune away \( p_2 \). However, since \( p_1 \) and \( p_2 \) are not in the same equivalence class because of the second test case, the search does not prune away \( p_2 \). Figure 2(a) illustrates that the additional test case splits programs \( s_1 \leadsto c_1 \) into two equivalence classes \( s_2 \leadsto c_2 \) and \( s_2 \leadsto d_2 \), so the search has to traverse the same subgraphs at \( c_2 \) and \( d_2 \) separately, to find out that both of them cannot reach \( t_2 \).

**Lens Algorithm** Our enumerative search does not have the aforementioned inefficiencies. It does not restart the search and uses just enough test cases to prune the search space. More specifically, when a counterexample is found, we build a new search graph according to the next test case only on the programs that pass all previous test cases, as shown in Figure 2(b). The search graph of test case 2 only includes programs that pass test case 1 (the highlighted paths in the search graph of test case 1). Therefore, we never revisit programs from \( s_1 \) to \( c_1 \). Conceptually, when we find a counterexample, we refine the search by only subdividing the feasible equivalence class, as shown in Figure 3(b).

Additionally, we discover that when we search for a program of length \( d \), we can in fact direct the search to a feasible equivalence class without constructing the other equivalence classes of programs of size \( d \), as shown in Figure 3(c). This can be achieved through bidirectional search, which builds the search graph from both \( s \) and \( t \), as shown in Figure 2(c).

### 2.2 Context-aware window decomposition

A context-aware window decomposition scales search techniques that can solve relatively small problems to larger problems without losing much optimality of the final solutions. The key idea is to inform the superoptimizer about the precise precondition and postcondition under which the optimized fragment will be executed. We harvest a precondition and postcondition from a context—code surrounding the code to be optimized—and used them to relax the correctness condition. The decomposition selects a random code fragment \( p_{spec} \) in a reference program and optimizes the fragment in the context of the prefix \( p_{pre} \) and the postfix \( p_{post} \) (as depicted in Figure 1). This process repeats until none of the fragments in the program can be optimized further. Consequently, this decomposition increases the effective size of programs that the superoptimizer can synthesize.

### 2.3 Cooperative search

A cooperative superoptimizer runs multiple search instances of enumerative, stochastic, and symbolic search. The superoptimizer exploits the strengths of all search techniques through communication between search instances, exchanging the best programs they have discovered so far.

To demonstrate the effectiveness of the cooperative superoptimizer, we show how it optimized a GA code fragment from a SHA-256 program. According to Figure 4(a), the cooperative superoptimizer was the only superoptimizer that found the best known code, while being as quick as the stochastic superoptimizer. Although some of the other techniques might seem better at the beginning, the cooperative superoptimizer eventually found the best solution that the other techniques could not; the cooperation costs some over-

![Figure 4. Optimizing a sequence of GA instructions from a SHA-256 program. 'stoch_s' is stochastic search that starts from random programs. 'stoch_o' is stochastic search that starts from the correct reference program.](image)
head but eventually pays off. Note that all superoptimizers execute the same number of search instances. The detailed descriptions of these five superoptimizers are in Section 7.3.

Figure 4(b) depicts how the cooperative superoptimizer arrived at the best solution. A stochastic instance that started mutating from the correct reference program first found a better solution, so it updated the best program shared between the search instances. An enumerative instance took that newly updated program, applied the context-aware window decomposition, and found two better solutions before another enumerative search instance took the latest best program, applied window decomposition, and found the final best program. Our experiment shows that the cooperative superoptimizer increased the number of benchmarks in which the superoptimizer found best known solutions consistently from 23 to 29 (out of 32) over using the enumerative search alone. We define a superoptimizer as consistent at solving a benchmark if it found best known solutions in all runs.

3. The LENS Algorithm

In Section 2, we outlined the LENS algorithm’s pruning strategies. For the sake of simplification, we assumed that the size of the synthesized program was fixed a priori. The complete description provided in this section explains how the algorithm simultaneously grows the program size and refines the search.

3.1 Representation of Search Graphs

Each test case \((i_k, o_k)\) is associated with a forward search graph \(F_k\) of program prefixes of length \(\ell_F\), and a backward search graph \(B_k\) of program postfixes of length \(\ell_B\). The root \(s_k\) of \(F_k\) is labeled with the input \(i_k\), and the root \(t_k\) of \(B_k\) is labeled with the output \(o_k\). We store \(F_1, \ldots, F_n\) in the nested map \(M_F\) such that \(M_F[u_1][u_2][\ldots][u_n]\) returns the set of programs \(p\) of length \(\ell_F\) such that \(p(i_1) = u_1, p(i_2) = u_2, \ldots, p(i_n) = u_n\). For example, in the search graphs in Figure 2(c), \(M_F[[1, 4]][[3, 2]]\) maps to three programs: (1) \(\text{sub r1, r0, r1, 1, s1} l\); \(\text{clz r0, r0, 0, 2} \text{clz r1, r1; clz r0, r0, and 3} \text{clz r1, r1}\). We use \(\text{Progs}(M_F)\) to refer to all programs stored inside \(M_F\).

The backward search graphs are stored differently, but to simplify the explanation of our algorithm, let us assume that the backward search graph offers the same interface; there is a map \(M_B\) such that \(M_B[u_1][u_2][\ldots][u_n]\) returns the set of programs \(p\) of length \(\ell_B\) such that \(p(u_1) = o_1, p(u_2) = o_2, \ldots, p(u_n) = o_n\). Our efficient implementation of the backward search graphs is described in Section 3.3.3.

3.2 The Algorithm

Algorithm 1 displays our main algorithm. We first create one test case. Therefore, at the beginning, we start the search from \(F_1\) containing only \(s_1\), and \(B_1\) containing only \(t_1\). Then, the main loop performs two actions—search and expand—in each iteration. The search phase searches for programs of size \(\ell_F\ell_B + 1\) that pass all test cases. When the search phase is complete, the expand phase increases the size of programs we will be searching in the next iteration by one. This process repeats until timeout.

The expanding phase (on line 10–13) increases the size of programs by expanding all leaf nodes of either \(F_1\) or \(B_1\). \(\text{Forward}\) is a heuristic function that decides whether to expand forward or backward. In particular, we expand each leaf node \(u\) in \(F_1\) by adding \(u \xrightarrow{\text{inst}} v\) for all \(\text{inst} \in \text{Insts}\), where \(\text{Insts}\) is a set of all possible instructions. Similarly, we expand each leaf node \(v\) in \(B_1\) backward by adding \(u \xrightarrow{\text{inst}} v\) for all \(\text{inst} \in \text{Insts}\).

The searching phase (on line 7–9) find programs that pass all \(n\) test cases by finding an instruction that can connect

---

**Algorithm 1 Main search**

1: \(n \leftarrow 1\)
2: \(\ell_F \leftarrow 0, \ell_B \leftarrow 0\)
3: \(p_{\text{spec}} \leftarrow \text{ReduceBitwidth}(p_{\text{spec}})\)
4: \(\text{cost} \leftarrow \text{cost}(p_{\text{spec}})\)
5: \((I, O) \leftarrow \text{GenTest}(p_{\text{spec}})\)
6: \(M_F \leftarrow \text{Init}(I), M_B \leftarrow \text{Goal}(O)\)
7: \(\text{while true do}\)
8: \(\text{for all inst} \in \text{Insts} \text{ do } \text{Searching Phase}\)
9: \(\quad (M_F, M_B) \leftarrow \text{Connect}(M_F, M_B, \text{inst}, 1)\)
10: \(\text{if Forward?(}\ell_F, \ell_B)\text{ then } \text{Expanding Phase}\)
11: \(\quad M_F \leftarrow \text{ExpandForward}(M_F), \ell_F \leftarrow \ell_F + 1\)
12: \(\text{else}\)
13: \(\quad M_B \leftarrow \text{ExpandBackward}(M_B), \ell_B \leftarrow \ell_B + 1\)

---

**Algorithm 2 Connect and refine**

Global variables: \(I, O, \text{cost}, n, p_{\text{spec}}, p_{\text{spec}}\)

1: \(\text{function } \text{CONNECT}(M_F, M_B, \text{inst}, k)\)
2: \(\text{if } k > n \text{ then } \text{Pass all test cases}\)
3: \(\text{for all } p \in M_F, p' \in M_B \text{ do } \text{ searchable programs }\)
4: \(\quad \text{if cost}(p \oplus \text{inst} \oplus p') < \text{cost then}\)
5: \(\quad \text{Verify}(p \oplus \text{inst} \oplus p')\)
6: \(\text{Build search graph on test case } k\)
7: \(\text{if } M_F \text{ is not a map then } \text{M_F is a set of programs}\)
8: \(\quad M_F \leftarrow \text{BuildForward}(M_F, I[k])\)
9: \(\text{if } M_B \text{ is not a map then } \text{M_B is a set of programs}\)
10: \(\quad M_B \leftarrow \text{BuildBackward}(M_B, O[k])\)
11: \(\text{End}\)
12: \(\text{for all } u \in \text{keys}(M_F) \text{ do } \text{Search for a connection}\)
13: \(\quad v \leftarrow \text{inst}(u)\)
14: \(\text{if } v \in \text{keys}(M_B) \text{ then } \text{Find a connection, so refine the search}\)
15: \(\quad (M_F[u], M_B[v]) \leftarrow \text{Connect}(M_F[u], M_B[v], \text{inst}, k + 1)\)
16: \(\text{return } (M_F, M_B)\)
17: \(\text{function } \text{VERIFY}(\hat{p})\)
18: \(\text{if } \hat{p} \equiv p_{\text{spec}} \text{ then } \text{Check via a constraint solver}\)
19: \(\text{for all } p \in \text{IncreaseBitwidth}(\hat{p}) \text{ do } \text{Found a better program!}\)
20: \(\text{if } p \equiv p_{\text{spec}} \text{ then } \text{cost} \leftarrow \text{cost}(p)\)
21: \(\text{yield } p\)
22: \(\text{else}\)
23: \(\quad n \leftarrow n + 1\)
24: \(\quad (I[n], O[n]) \leftarrow \text{CounterExample}(p_{\text{spec}}, \hat{p})\)
leaf nodes in $F_1, \ldots, F_n$ to leaf nodes in $B_1, \ldots, B_n$ respectively. The main algorithm calls $\text{Connect}$ to find such programs. $\text{Connect}(M_F, M_B, \text{inst}, k)$, shown in Algorithm 2, searches for programs in $\text{Progs}(M_F) \oplus \text{inst} \oplus \text{Progs}(M_B)$ that pass test cases $k$ to $n$. It maintains the invariant that all programs in $\text{Progs}(M_F) \oplus \text{inst} \oplus \text{Progs}(M_B)$ pass all test cases $1$ to $k - 1$. This invariant is the key to refining the search only on a feasible equivalence class.

After $F_k$ and $B_k$ are built, the loop on lines 12–15 searches for a leaf node $u$ in $F_k$ and $v$ in $B_k$ that can be connected by $\text{inst}. \text{keys}(M_F)$ and $\text{keys}(M_B)$ on line 12 and 14 are sets of leaf nodes in $F_k$ and $B_k$. If $\text{inst}$ can connect $u$ to $v$, programs in $\text{Progs}(M_F[u]) \oplus \text{inst} \oplus \text{Progs}(M_B[v])$ pass test case $k$, so the algorithm refines the search on $\text{Progs}(M_F[u]) \oplus \text{inst} \oplus \text{Progs}(M_B[v])$ with the next test case $k + 1$. For our running example in Figure 2(c), we find an instruction $\text{sub} r0, r0, r1$ connecting $u_1$ and $v_1$ of test case 1, so we refine the search on the highlighted programs $s_1 \rightsquigarrow u_1 \rightarrow v_1 \rightsquigarrow t_1$.

When we recursively call $\text{Connect}$, $M_F$ will eventually become a set of programs instead of a nested map, as well as $M_B$. Lines 7–10 take care of building $F_k$ for programs in $M_F$ and $B_k$ for programs in $M_B$. $F_k$ and $B_k$ for each $k$ are built once and saved on line 15 to be used later when $\text{Connect}$ is called with different $\text{insts}$. If there are no more test case left, lines 2–5 verify all programs in $M_F \oplus \text{inst} \oplus M_B$ against the reference program. $\text{Verify}$ function performs equivalence verification. If the two programs are not equivalent, an counterexample is added to $I$ and $O$ on line 25. If they are equivalent, the algorithm yields the candidate program and continues searching for solutions with lower costs until timeout.

### 3.3 Implementation Details

#### 3.3.1 Challenges of Backward Search

We have identified two main challenges in implementing backward search in a program synthesizer. First, the synthesizer needs to evaluate an instruction backward; it needs an inverse function for every instruction. Second, in the forward direction, an instruction $\text{inst}$ is a one-to-one function that map a state $u$ to $v$. In contrast, in the backward direction, $\text{inst}$ is a one-to-many function that map the state $v$ to a set of states, one of which is $u$.

Fortunately, we can mitigate these challenges by reducing bitwidth, using only four bits to represent a value. First, we can avoid implementing an inverse emulator by constructing an inverse function table for every instruction. We execute every instruction on all possible combinations of 4-bit input arguments’ values and memorize them in the inverse table. Second, the small bitwidth also reduces the number of states an instruction can transition to in the backward direction. For example, in 32-bit domain, an inverse instruction $\text{add}$ transitions from one state to $2^{32}$ states; in contrast, in 4-bit domain, the same instruction only transitions to $2^4$ states.

### 3.3.2 Reduced Bitwidth

Let $\text{bit}$ be the actual bitwidth and $\hat{\text{bit}}$ be the reduced bitwidth, which is four in our case. The reduced bitwidth not only enables the backward search but also allows us to initially divide the search space more coarsely, which is desirable because the search graph even for a single test can be very large. For example, an ISA with four 32-bit registers can have $2^{32 \times 4}$ states and, hence, up to $2^{32 \times 4}$ nodes in the graph.

Apart from the second-step equivalence verification (line 20 of Algorithm 2), the search algorithm operates in the reduce-bitwidth domain. Therefore, we need both reduced-bitwidth and precise versions of a program state and an ISA emulator. We implement an emulator that can be parameterized by bitwidth to instantiate both versions. For example, the precise ARM emulator interprets instruction $\text{movt} r0, 1$ as writing 1 to the top 16 bits of a 32-bit register. The 4-bit ARM emulator should interpret the same instruction as writing 1 to the top 2 bits of a 4-bit register. Implementing a parameterizable program state is simple. We just need to use a specified number of bits to represent each entry in a program state.

Additionally, we must have a way to convert programs between the two domains. In particular, at the beginning, we convert the reference program $p_{\text{ref}}$ from the precise domain to the reduced-bitwidth domain (line 3 in Algorithm 1) by replacing constants appearing in the program with their reduced-bitwidth counterparts. We replace a constant $c$ using the following function $\alpha$:

$$\hat{c} = \alpha(c) = \begin{cases} 
\text{bit} & \text{if } \text{shift}?(c) \land (c = \text{bit}) \\
\text{bit} - 1 & \text{if } \text{shift}?(c) \land (c = \text{bit} - 1) \\
\frac{\text{bit}}{2} & \text{if } \text{shift}?(c) \land (\text{bit}/2 \leq c < \text{bit} - 1) \\
1 & \text{if } \text{shift}?(c) \land (1 < c < \text{bit}/2) \\
c \mod 2^{\text{bit}} & \text{otherwise}
\end{cases}$$

where $\text{shift}?(c)$ checks if $c$ is a shift operand. $\alpha$ is designed to preserve semantics of shift operations in a meaningful way. For example, it translates shift by 31 in 32-bit domain to shift by 3 in 4-bit domain. Apart from shift constants, $\alpha$ simply masks in the lowest $\hat{\text{bit}}$ bits.

During this conversion, we memorize every replacement of $c$ with $\hat{c}$, so that we can map each reduced-bitwidth constants back to the set of original constants to obtain candidate programs in the precise domain. We construct the replacement map $\gamma$ by storing $\gamma[\hat{c}] \leftarrow \gamma[\hat{c}] \cup \{c\}$ for every constant $c$ in $p_{\text{ref}}$. Before the precise equivalence verification, the reduced-bitwidth constant $\hat{c}$ is replaced with every constant in the set $\gamma[\hat{c}]$ (line 19 in Algorithm 2) with the expectation that one of them will lead to a correct solution.

We are able to optimize many bitwidth-sensitive programs (e.g. population count and computing higher-order half of multiplication) using this reduced-bitwidth trick.

#### 3.3.3 Data Structure for Backward Search Graph

We could store backward search graphs the same way we store forward search graphs. However, it would require a
large amount of memory because in the backward direction, an instruction is a one-to-many function; one program post-
fix can appear in a large number of backward equivalence classes. Instead of using a nested map to store all backward
search graphs, we construct \(n\) separate maps to store \(n\) back-
ward search graphs \(B_1, \ldots, B_n\). We can find a program post-
fix \(p\) such that \(p(u_1) = o_1, \ldots, p(u_n) = o_n\), by looking up
\(Y[u_1] \cap \ldots \cap Y[u_n]\). The pseudocode in Algorithm 1 and
Algorithm 2 has to be modified slightly to support this data
structure.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Non-context-aware</th>
<th>Context-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence verification</td>
<td>(p_{spec} \equiv p)</td>
<td>(p_{pre} \oplus p_{spec} \oplus p_{post} \equiv p_{pre} \oplus p \oplus p_{post})</td>
</tr>
<tr>
<td>Input text-cases update</td>
<td>(I = I \cup {i_{ce}})</td>
<td>(I = I \cup {p_{pre}(i_{ce})})</td>
</tr>
<tr>
<td>Output text-cases update</td>
<td>(O = O \cup {p_{spec}(i_{ce})})</td>
<td>(O = O \cup {p_{pre} \oplus p_{spec}(i_{ce})})</td>
</tr>
</tbody>
</table>

Table 1. The differences between non-context-aware and context-aware decomposition. \(p\) is a candidate program. \(i_{ce}\) is the input counterexample returned by the constraint solver if the candidate program is not equivalent to the reference program.

4. Context-Aware Window Decomposition

We can scale a search technique that can synthesize rel-
atively small programs to optimize larger programs using
a decomposition. Let \(p_{ref}\) be a large program to be opti-
mized, and \(L\) be a window size. We can decompose \(p_{ref}\) into \(p_{pre} \oplus p_{spec} \oplus p_{post}\) such that \(\text{length}(p_{spec}) < L\), and
optimize \(p_{spec}\), the code inside the window. Peephole opti-
mizations will try to optimized \(p_{spec}\) alone, or in the best
scenario, with a precondition that is often not precise. The
precondition and postcondition relax the correctness con-
dition and provide invariants that may be exploited by the
search. Therefore, we believe that optimizing \(p_{spec}\) with the
most precise precondition and postcondition, essentially in
the context of its prefix \(p_{pre}\) and postfix \(p_{post}\), can lead
to finding a better program. We call this decomposition a
context-aware window decomposition. In our implementa-
tion, we pick a random position of the window and optimize
the program. This process repeats until we cannot optimize
the program at any window’s position anymore.

To support the context-aware decomposition, we need to
modify search algorithms slightly. Note that any search
technique can be modified to be context-aware. Recall that a
search technique looks for a program \(p\) such that for each \(i \in I, o \in O, p(i) = o\). To make the search context-aware, we
do not need to change this search routine, but only need
to adjust the equivalence condition used during equivalence
verification and the way test cases are updated as shown in
Table 1. Normally, when we find \(p\) that passes all test cases,
we use a constraint solver to verify if \(p_{spec} \equiv p\). If they
are not equivalent, the constraint solver will return an input
counterexample \(i_{ce}\), which we use to update the test inputs \(I\)
and test outputs \(O\) as shown in Column ‘non-context-aware’.

Then, the search continues to find a new candidate program,
and so on. To make the search context-aware, we ask the
constraint solver if \(p\) is equivalent to \(p_{spec}\) in the context
of \(p_{pre}\) and \(p_{post}\), in particular if \(p_{pre} \oplus p_{spec} \oplus p_{post} \equiv
p_{pre} \oplus p \oplus p_{post}\). If they are not equivalent, the constraint
solver will return \(i_{ce}\), which is an input to \(p_{pre}\) (not directly
to \(p\)), so we have to update the test cases differently as shown in
Column ‘context-aware’.

Concrete Example

Assume we want to optimize the following ARM program:

\[
P_{pre}: \text{cmp } r3, r4 \\
\text{moveq } r1, #0 // \text{mov if } r3 = r4 \\
\text{movne } r1, #1 // \text{mov if } r3 \neq r4 \\
P_{spec}: \text{cmp } r2, #31 \\
\text{movbi } r1, #0 // \text{mov if } r2 > 31 \\
\text{andis } r1, r1, #1 // \text{and if } r2 < 31
\]

The decomposition selects the window as labeled; \(p_{post}\) is
empty. Without \(p_{pre}, p_{spec}\) cannot be improved because no
faster code modifies \(r1\) as \(p_{spec}\) does. With \(p_{pre}\), however,
the superoptimizer learns that the value of \(r1\) is either 0
or 1 at the beginning of \(p_{spec}\), so the last instruction \(r1 =
(1 \& 1) \neq 0\) does not have any effect. Thus, the superoptimizer
can simply remove it. Note that we do not have to explicitly
infer this precondition of \(p_{spec}\). It is implicit, captured by
running \(p_{pre}\) along with \(p_{spec}\) during test case evaluations
and equivalence verification. We also find that \(p_{post}\) helps
the superoptimizer discover faster code.

5. Cooperative Superoptimizer

To utilize strengths of different search techniques, we intro-
duce a cooperative superoptimizer that combines all search
techniques in a simple fashion. The cooperative superopti-
mizer launches all search techniques in parallel and may run
more than one search instance of each search technique.

5.1 Terminology

This section defines the terminology and symbols of vari-
ations of different search techniques used in the rest of the pa-
per. The base search algorithms are symbolic (\(SM\)), stochastic
(\(ST\)), and enumerative (\(E\)). There are two modes of
search. Synthesize mode (\(s\)) is when a search does not use a
starting correct program except for equivalence verification.
Optimize mode (\(o\)) is when a search uses a starting correct
program beyond equivalence verification. The table below
summarizes the symbols we use.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E^s)</td>
<td>enumerative on entire code fragment</td>
</tr>
<tr>
<td>(E^o)</td>
<td>enumerative with decomposition</td>
</tr>
<tr>
<td>(SM^s)</td>
<td>symbolic on entire code fragment</td>
</tr>
<tr>
<td>(SM^o)</td>
<td>symbolic with decomposition</td>
</tr>
<tr>
<td>(ST^s)</td>
<td>stochastic that starts from a random program</td>
</tr>
<tr>
<td>(ST^o)</td>
<td>stochastic that starts from the input correct program</td>
</tr>
</tbody>
</table>
We develop an ISA-specific constant to our superoptimizer, and (ii) the symbolic search for that particular ISA. We use four sizes of window lengths: 2, 3, 4, and 7. We instantiate GreenThumb to build superoptimizers for two very different ISAs.

**ARM** is a RISC architecture that is widely used in many devices. We implement a superoptimizer for ARMv7-A and model the performance cost function based on ARM Cortex-A9 [4]. An ARM program state includes 32-bit registers, memory, and condition flags. The default instruction representation provided by GreenThumb, which includes opcode and operands, is extended to support ARM instructions by including a condition code suffix and optional shift. We also extend GreenThumb’s stochastic search by adding new mutation rules: mutating condition code, and mutating optional shift. The smallest window size \(L\) is set to 2. Recall that there are four sizes of window \(L, 2L, 3L, \) and \(4L\).

**GreenArrays** GA144 is a low-power processor, composed of many small cores [12]. It is a stack-based 18-bit processor. Each core has two registers, two small stacks, and memory. Each core can communicate with its neighbors using read and write instructions. The program state for GA includes registers, stacks, memory, and a communication channel, similar to the one used in the superoptimizer in Chlorophyll [20]. A communication channel is an ordered list of (data, neighbor port, read/write) tuples representing the data that the core receives and sends. For two programs to be equivalent, their communication channels have to be identical. Only one GA instruction, fetch-immediate, has an operand, which is for specifying the immediate constant value, so we extend the stochastic search to mutate an operand only for that opcode. We set the smallest window size \(L\) to 7. We model the performance cost based on GA144’s instruction timing [12].

**Limitations**

We do not model memory access latency variations caused by misses at different levels of caches. We assign the same cost to all loads and stores. Therefore, our performance model is imprecise; as a result the superoptimizer may output a program that is actually slower than other candidates it has explored. To work around this problem, the superoptimizer can output the best ten programs instead of only the best one. This way, we can try running all of them on the real machine and select the fastest one empirically.

The second limitation is that the superoptimizer can only optimize code without loops and branches. In order to optimize across multiple basic blocks with loops and branches, we will need to modify the superoptimizer.

### 5.2 Communication between Search Instances

The search instances aid each other by exchanging information about the current best solution equivalent to \(p_{\text{ref}}\). When a search instance finds a new best program, it updates the shared best solution \(p_{\text{best}}\). The other search instances may obtain \(p_{\text{best}}\) to aid in their search processes. In particular, different types of search techniques utilize \(p_{\text{best}}\) as follows:

- \(E^a\) and \(SM^a\) do not use \(p_{\text{best}}\).
- \(E^o\) and \(SM^o\) apply the context-aware window decomposition on \(p_{\text{best}}\).
- \(ST^a\) reduce its search space by only exploring programs with up to \(\text{length}(p_{\text{best}})\) instructions.
- \(ST^o\) restarts the search from \(p_{\text{best}}\). In practice, it is better to allow some divergence among stochastic instances. Therefore, our stochastic instances check \(p_{\text{best}}\) every 10,000 mutations and restart the search from \(p_{\text{best}}\) only if \(\text{cost}(p_{\text{best}})\) is much less than the cost of the local best solution; in our implementation, we restart when the difference is more than 5.

### 5.3 Practical Configuration of Search Instances

We present a configuration for allocating search instances that worked well in our experiment; however, it might not be optimal. Our cooperative superoptimizer executes \(N\) search instances with the following distributions: \(N/2 - 1\) \(E^o\), one \(E^a\), two \(ST^a\), three \(ST^o\), and the rest for \(SM^o\). We dedicate almost half of the threads for enumerative search because it performs the best in most benchmarks (see Section 7.1). Multiple enumerative instances attempt to optimize different parts of the program at the same time, reducing overall search time to find a final solution. We allocate one thread for \(E^a\) because if the size of the final solution is small, \(E^a\) will find an optimal solution quickly. A few \(ST^o\) instances are allocated because they can perform very well on some benchmarks on which \(E\) performs poorly. We also dedicate a few threads for \(ST^o\) instances because they often help \(E^o\) instances reach the final solution faster. Finally, we allocate the rest of the resources to \(SM^o\), which helps discover optimizations that involve synthesizing arbitrary constants.

For search instances that use the window decomposition, we use four sizes of window \(L, 2L, 3L, \) and \(4L\), where \(L\) is a constant specific to the ISA.

### 6. Implementation

We develop GreenThumb [21], a framework for building superoptimizers for different ISAs and testing search techniques. We define the semantics of an ISA by implementing an ISA emulator in Rosette [30], which is built on top of Racket. The enumerative and stochastic search uses the emulator to execute a sequence of instructions on concrete program states. Additionally, since the emulator is written in Rosette, we also obtain, for free, (i) the program equivalence verifier, and (ii) the symbolic search for that particular ISA.

We instantiate GreenThumb to build superoptimizers for two very different ISAs.
faster on average) and obtained better solutions (the performance cost of our code is, on average, 18% lower than that of stochastic search).

We implemented all search techniques as well as ARM and GA emulators in our framework using Racket. Since all search techniques are implemented in the same language and using the same emulator, we can compare them fairly. As discussed in Section 7.5, we tried to make a fair comparison between all search techniques.

This section presents detailed evaluation of our algorithms, starting from the new enumerative algorithm, using the following benchmark suites.

**ARM Hacker’s Delight Benchmarks** consist of 16 of the 25 programs identified by [13] drawn from Hacker’s Delight [33]. We excluded the first nine programs from our set of benchmarks because they are very small. We used code produced by gcc -march=armv7-a -00 as the input programs to the superoptimizers. Their sizes ranged from 16 to 60 instructions. The timeout was set to one hour.

**GA Benchmarks** consist of frequently-executed basic blocks from MD5, SHA-256, FIR, sine, and cosine functions from the Chlorophyll compiler’s benchmarks [20]. We used Chlorophyll without superoptimization to generate these basic blocks. The sizes of the input programs in this benchmark suite ranged from 10 to 56 instructions. The timeout was set to 20 minutes.

### 7.1 Experiment I: Evaluating the LENS algorithm

Experiment I is designed to evaluate the base search techniques: $SM^*$, $E^*$, and $ST^*$. Recall that superscript * indicates synthesize mode (no window decomposition). This experiment will help us answer which search technique is a suitable building block for a superoptimizer with window decomposition. For each benchmark, we ran each search technique on a single thread 16 times.

**Hypothesis A** Enumerative search is faster and can solve larger benchmarks than the other base search techniques.

$E^*$ is superior in terms of speed and scalability: it was the fastest search; it solved all except two benchmarks; and it could solve larger benchmarks than other synthesizers. Regarding consistency—which is desirable because it obviates the need for redundant instances, improving the chances of finding optimal solutions—almost all of $E^*$ 16 search instances found optimal solutions in each of its solved benchmarks. Note that there is small amount of randomness in the enumerative search because the initial test cases are generated randomly.

There were a total of 22 benchmarks in which one of the search techniques found optimal solutions in at least one of the 16 runs. Columns $SM^*$, $E^*$, and $ST^*$ of Figure 5 summarize the results. Figure 5(a) displays the number of benchmarks solved by each search technique, categorized by size. A search technique solved a benchmark if it an optimal solution in one of its run. Row ‘benchmarks’ in Figure 5(b)

summarizes the numbers of solved benchmarks. Row ‘instances’ displays the average numbers of search instances that found optimal solutions per solved benchmark. In terms of search time, we evaluated each search technique against $E^*$ by comparing the best runs of the benchmarks they both solved. Row ‘$E^*$ speedup’ shows how much faster $E^*$ was on average compared to a particular search technique.

According to Figure 5(a), $E^*$ could synthesize larger ARM programs than $ST^*$ and $SM^*$ could. For GA, $E^*$ could synthesize larger programs than $SM^*$. While $E^*$ and $ST^*$ were comparable at synthesizing large GA programs, $ST^*$ was much worse at synthesizing smaller GA programs. This might be because the cost function of $ST^*$ does not fit well with these GA benchmarks, or the mutations we have are not the best for GA. Interestingly, the largest GA benchmark, which $E^*$ failed to solve, was solved by $ST^*$. This result suggests that sometimes the cost function can be very effective in guiding the search in some particular problems. Another benchmark that $E^*$ failed to solve can be solved by $SM^*$. This is because the optimal program contains a constant not included in the pre-defined constant list of $E^*$ and $ST^*$.

**Hypothesis B** The LENS algorithm improves on the existing enumerative algorithms.

With the same experimental setting, we compared multiple versions of enumerative search:

- $E^*$: LENS with all pruning strategies
- $E^{\prime\prime}$: $E^*$ without backward search (unidirectional search)
- $E^{\prime\prime\prime}$: $E^{\prime\prime}$ without reduced-bitwidth trick
- $E^{\prime\prime\prime\prime}$: $E^{\prime\prime\prime}$ without refinement through incremental test cases. $E^{\prime\prime\prime\prime}$ represents the existing enumerative search but without the stack representation [6].

Columns $E^*$–$E^{\prime\prime\prime\prime}$ of Figure 5 summarizes the results. The pruning strategies we introduce not only increase the size of code an enumerative search can solve but also speed up the search: $E^*$ was, on average, 11x faster than $E^{\prime\prime\prime\prime}$.
7.2 Experiment II: Evaluating window decomposition

Experiment II is designed to test the effectiveness of context-aware window decomposition. We test $E^o$, which is context-aware, against a modified version of $E^o$, which is not context-aware, on the 12 benchmarks that $E^o$ cannot synthesize optimal solutions from the previous experiment. Recall that the superscript $^o$ indicates optimize mode (see Section 5.1). On ARM benchmarks, we ran a superoptimizer using 32 $E^o$ search instances on a 16-core hyper-threaded machine. On GA benchmarks, we ran 16 search instances on a 16-core Amazon EC2 machine. For each benchmark, we repeated the experiment three times.

Hypothesis C The context-aware window decomposition technique enables the enumerative search to find better code than does the non-context-aware window decomposition.

Considering the best out of the three runs, in six benchmarks, the context-aware decomposition found solutions with 1.3x–3x lower cost than did the non-context-aware decomposition. In the rest, both of them found solutions with the same costs.

7.3 Experiment III: Evaluating cooperative search

Experiment III is designed to evaluate superoptimizers based on different search techniques with context-aware window decomposition. We use the same experimental set up as in Section 7.2. We evaluate the following five versions of superoptimizers, each of which runs $N$ search instances ($N = 32$ for ARM, and $N = 16$ for GA).

<table>
<thead>
<tr>
<th>Superoptimizer</th>
<th>Search instances used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ST^*$</td>
<td>all $ST^o$ instances with no communication</td>
</tr>
<tr>
<td>$ST^\prime$</td>
<td>all $ST^o$ instances with no communication</td>
</tr>
<tr>
<td>$SM$</td>
<td>one $SM^o$ instance, $N - 1 SM^o$ instances</td>
</tr>
<tr>
<td>$E$</td>
<td>one $E^o$ instance, $N - 1 E^o$ instances</td>
</tr>
<tr>
<td>$C$</td>
<td>one $E^o$, $N/2 - 1 E^o$, two $ST^o$, three $ST^o$, and the rest for $SM^o$ instances</td>
</tr>
</tbody>
</table>

Search instances of each superoptimizer communicate with each other except in $ST^*$ and $ST^\prime$, which represent STOKE implemented in our framework. In $E$, $SM$, and $C$, we add one instance of an enumerative or symbolic search in synthesize mode ($E^o$ or $SM^o$) because these instances can find an optimal solution should the optimal solution be small.

Hypothesis D The enumerative superoptimizer can often synthesize best known programs more consistently and faster than the stochastic and symbolic superoptimizers.

$E$ was consistent on 2.1x, 2.6x, and 1.4x more benchmarks than $ST^*$, $ST^\prime$, and $SM$. We define a superoptimizer as consistent at solving a benchmark if it found programs as optimal as the best known solution in all runs. Consistency is desirable because in practice we want to find the best program in one run not multiple runs. Then, we did a pair-wise comparison of the median search time between $E$ and each of the other superoptimizers on the benchmarks they both solved consistently. We found that $E$ was also on average 9x, 4.6x, and 14x faster than $ST^*$, $ST^\prime$, and $SM$.

Hypothesis E The cooperative superoptimizer improves on the enumerative superoptimizer by utilizing the strengths of other search techniques.

We compare $C$ and $E$. While $E$ uses only enumerative search instances, $C$ uses enumerative as well as symbolic and stochastic search instances. According to the result, $C$ was consistent at finding best known solutions on 29 out of 32 benchmarks, while $E$ was consistent on 23 benchmarks. $C$ and $E$ were comparable in term of search time; $C$ was 33% faster, on average. Columns $\tilde{C}$ of Figure 6 and Table 2 display the costs of the best correct programs found by $C$ and its median time to find the best known solutions for all benchmarks. Compared to the algorithm used in the state-of-the-art superoptimizer (STOKE), $\tilde{C}$ was, on average, 12x faster than the best of $ST^*$ and $ST^\prime$. The performance cost of code produced by $\tilde{C}$ is, on average, 18% lower than that of the best from $ST^*$ and $ST^\prime$.

We also tested $C'''$—the cooperative superoptimizer with the enumerative search without our pruning strategies—to examine how much the performance of the enumerative instances affect the performance of the cooperative superoptimizer. Columns $\tilde{C}'''$ of Figure 6 and Table 2 display the costs of the best correct programs found by $\tilde{C}'''$ and its median time to find the best known solutions. According to the result, $\tilde{C}'''$ could not consistently solve seven benchmarks that $\tilde{C}$ could. Hence, we conclude that our pruning strategies in the enumerative search are crucial for obtaining the best performance out of the cooperative superoptimizer.

7.4 Experiment IV: Runtime speedup over gcc -O3

Experiment IV is designed to test the effectiveness of the cooperative superoptimizer against an optimizing compiler. We measure the execution time of all benchmarks in this experiment on an actual ARM Cortex-A9.

Hypothesis F Cooperative superoptimizer can optimize code generated from a non-optimizing compiler and obtain code as fast as generated from an optimizing compiler.

From the experiment in Section 7.3, $C$ optimized code generated from gcc -O0 and produced code as fast as gcc -O3 code for all ARM benchmarks. In fact, $\tilde{C}$ found faster code than those generated from gcc -O3 on five benchmarks. One of them is 17.8x faster. Thus, for the new architectures for which we do not have good optimizing compilers, our superoptimizer can help generating efficient code.
Figure 6. Costs of best programs found by the different superoptimizers (normalized by the cost of the best known program). A dash represents the cost of the best program found in one run. A dash may represent more than one run if the best programs found in different runs have the same cost. If one or two runs did not find any correct program that is better than the input program, the vertical line is extended past the chart. If none of the runs found a correct program that is better than the input program, a rectangle is placed at the top of the chart.

Table 2. Median time in seconds to reach best known programs. "-" indicates that the superoptimizer failed to find a best known program in one or more runs. Bold denotes the fastest superoptimizer to find a best known program in each benchmark.

Table 3 summarizes characteristics of the program fragments found by $C$ that are faster than those generated by gcc -O3. Column ‘runtime speedup’ reports how much faster the fragments are when running on an actual ARM processor. The last column demonstrates that different base search techniques contribute to finding the best solutions in many benchmarks. For example, in the wi-txrate5a benchmark from WiBench’s rate matcher kernel, a $SM^o$ instance first optimized the input program, and then a $ST^o$ instance optimized the program found by the $SM^o$ instance and arrived at the best known solution. In mi-bitshift from MiBench’s
Bit shift benchmark, an $E^o$ instance immediately found the best program by applying the optimization explained in the concrete example in Section 4. In p21 from Hacker’s Delight, the path to the best known solution involves passing the latest best programs between many $E^o$, $SM^o$, and $ST^o$ instances repeatedly.

To illustrate how the different types of search instances work together in practice, we explain how the cooperative superoptimizer found the best program in the mi-bitarray benchmark, getting a specific bit in an array. The superoptimizer found two optimizations (Optimization I and II) displayed in Figure 7. The code in blue is inside a window, and the rest are the context used in the window decomposition. First, a $SM^o$ instance optimized code inside a small window of two instruction to one instruction. The $SM^o$ instance was able to perform this optimization because it can synthesize an arbitrary constant, in this case, 248. After the $SM^o$ instance discovered Optimization I, an $E^o$ instance optimized the code further. Optimization II performed by the $E^o$ instance, in fact, consists of two different optimizations. The first optimization transforms:

```
cmp r1, #0
add r2, r1, #7
movge r2, r1
```

eliminating the `cmp` instruction and the conditional suffix. Note that this transformation is valid in any context. In the second optimization, the superoptimizer learned from the postfix—specifically at the instruction `sub r3, r1, r3`—that only the difference between the values of $r1$ and $r3$ matters, and the exact values of $r1$ and $r3$ do not. This particular optimization illustrates that not only precondition but post-condition also helps the superoptimizer discover more optimizations. Notice that the $E^o$ instance used the constant 248, found by $SM^o$, to synthesize the final code, as the optimized fragment contains 248. Hence, in order to obtain the final code in this benchmark, we need the enumerative search, the symbolic search, and the context-aware window decomposition all together.

### 7.5 Existing Superoptimizers’ Implementations

The original stochastic superoptimizer (STOKE) [24, 25] is for x86. Consequently, we could not use STOKE in our experiments. STOKE can evaluate approximately 10^6 candidates per second by executing programs natively [25] or running emulators on a cluster of machines [24]. Without an ability to run programs natively or an access to run programs on a cluster of machines, one will not be able to achieve this kind of performance. Nevertheless, our stochastic superoptimizer is able to evaluate approximately 20,000 candidates per second and synthesize up to 10 ARM instructions within an hour using emulators on one machine with 32 cores. However, the optimized program with 10 instructions that our stochastic superoptimizer synthesized is not optimal, so it is not reported in the experiment in Section 7.1. Note that the size of our ARM search space is similar to x86 search space explored in the original STOKE without JIT [24]: they both have 400–1,000 different variations of opcodes [26]. Although ARM has much fewer actual opcodes than x86, many variations are created by the combination of opcodes, optional shift, and conditional suffixes.
Similarly, we created $E^{xtr}$ by modifying $E^x$ to implement the algorithm used in the SIMD synthesizer [6] and the SyGus enumerative solver [3]. We note that $E^{xtr}$ does not use the stack-based program representation, used in [6] to remove search space symmetries due to register renaming. We did not use this representation because we observed that some optimal programs cannot be obtained from this representation, unless we introduce new pseudo-instructions for peaking into the stack and dropping values from the stack. Note that this optimization is orthogonal from the search algorithm and can improve our search technique.

8. Related Work

**Symbolic search** is popular in program synthesis tools such as Sketch [27] and Rosette [31]. This search technique is also used in the Chlorophyll compiler’s superoptimizer [20]. Although constraint solvers have many clever pruning strategies (e.g. conflict clauses) and heuristics to make decisions, constraint solvers are not optimized for program synthesis problems. Component-based synthesis [13] introduces an alternative encoding, which significantly improves the performance of a symbolic search; however, even with this encoding, the symbolic solver from SyGus’14 competition still did not perform well [3]. Another pruning strategy using divide-and-conquer to break QFBV formula potentially reduces synthesis time by many orders of magnitude [28], but it is likely synthesizing the same program as given. The refutation-based approach used in the CVC4 solver [23], the winner of SyGus’15 competition, is also not suitable for superoptimization problems because it tends to produce very large solutions with many if-else constructs.

**Stochastic search**, first used in STOKE [24, 25], randomly mutants a program to another using cost function to determine the acceptance of the mutation. STOKE is the first superoptimizer that is able to synthesize large programs (10–15 x86 instructions) in under an hour. The use of cost function to guide the search is one of the keys to its effectiveness. The weakness of this search is a possibility to get stuck at local minima and, as a result, fail to reach an optimal solution.

**Enumerative search** is used in many superoptimizers and program synthesizers. Enumerative search can be extremely fast if it is done right, as the winning teams of two synthesis competitions (ICFP’13 [2] and SyGus’14 [3]) employed this technique. This is because an enumerative search is highly customized to solve a particular problem it is designed for. The problem domain knowledge can be encoded into the algorithm used in the SIMD synthesizer [6] and the SyGus enumerative solver [3]. We note that $E^{xtr}$ does not use the stack-based program representation, used in [6] to remove search space symmetries due to register renaming. We did not use this representation because we observed that some optimal programs cannot be obtained from this representation, unless we introduce new pseudo-instructions for peaking into the stack and dropping values from the stack. Note that this optimization is orthogonal from the search algorithm and can improve our search technique.

9. Conclusion

This paper introduced the LENS algorithm, which can optimize larger program fragments compared to existing techniques. To optimize even larger program fragments, we applied a context-aware window decomposition, optimizing a subfragment of the entire code with the precise precondition and postcondition from the surrounding context. Lastly, we improved upon the LENS algorithm by combining symbolic and stochastic search into our system. To make superoptimization even more practical, we can cache superoptimized code to avoid an expensive search when optimizing programs we have seen before.

In summary, we introduced strategies to scale up superoptimization to optimize real-world programs. We hope that our work will enable program developers to use a superoptimizer to further optimize code generated from an optimizing compiler, when performance is critical. Similarly, we also hope to enable a rapid compiler construction for a new ISA by side stepping the laborious development of traditional compiler optimizations by using superoptimization.

Acknowledgments

This work is supported in part by Qualcomm Innovation Fellowship, MSR Fellowship, Grants from NSF (CCF–1139138, CCF–1337415, and ACI–1535191), U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers (FOA–0000619), and DARPA (FA8750–14–C–0011), as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.
References


