Superoptimizer Construction Framework with Efficient Hybrid Search

Phitchaya Phothilimthana
Aditya Thakur
Ras Bodik
Dinakar Dhurjati

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-14
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-14.html

April 10, 2015

Superoptimizer Construction Framework with Efficient Hybrid Search

Phitchaya Mangpo Phothilimthana* Aditya Thakur* Rastislav Bodik* Dinakar Dhurjati†

University of California, Berkeley* Qualcomm esearch†

{mangpo, athakur, bodik}@eecs.berekeley.edu inakard@qti.qualcomm.com

Abstract

A superoptimizer searches for an optimal implementation for a given input program in a target instruction set architecture (ISA). Despite its ability to generate optimal code, a superoptimizer is not commonly implemented for further optimizing code generated by a compiler. This is because building a superoptimizer for a new ISA requires a large amount of effort, and finding optimal code can be extremely slow if the search technique is inefficient.

We propose GREENTHUMB, an extensible framework for building superoptimizers. All that is required to extend GREENTHUMB to a new ISA is the implementation of an emulator for the new ISA. GREENTHUMB provides an efficient hybrid search technique that combines existing superoptimizer search techniques: symbolic search and mutation-based search. Additionally, we design a new correctness cost function (or fitness function), which is used in the mutation-based search, that leads to a more efficient search.

To illustrate the flexibility of the framework, we instantiate GREENTHUMB to two very different ISAs: ARM and GreenArrays. We evaluate the performance of the new hybrid search compared to the existing approaches in terms of speed and consistency of finding optimal solutions on a number of ARM and GreenArrays programs. We find that the hybrid search is the only search technique that finds an optimal program for all GreenArrays benchmarks, and the new cost function increases the number of runs in which the superoptimizer finds an optimal program by 20% on ARM benchmarks.

1. Introduction

Code optimization is more important today than ever before. A performance improvement of even a few percent can lead to significant cost saving for warehouse-scale data centers [26]. Optimizing always-running kernels can tremendously reduce energy consumption, which is crucial for battery-operated portable devices [17]. Code optimizers may reduce cost of devices by enabling developers to select lower-power computing resources and smaller memory [7].

Developing a code optimizer still remains a challenging problem. For example, although there exists a single ARM instruction that implements program `p25` [13], `gcc -O3` produces a program with 11 instructions (see § 8.2). The task of implementing a code optimizer is further exacerbated by the development of different instruction set architectures (ISAs) for different types of devices and processors. For example, ARM alone has over 30 variants of ISAs [36], and new architectures are constantly being developed [9, 10, 12, 21, 25, 33, 37].

A *superoptimizer*, first introduced by Massalin [20], generates an optimal implementation of a given input program in a target instruction set. Instead of relying on rewrite-based optimizations, superoptimizers *search* for a program that is correct and optimal given an optimality criterion. The superoptimization problem subsumes instruction selection, instruction scheduling, and local register allocation problems.

```
orr     r0, r1, r0   a!; push; -;
mvn     r0, r0       pop; - ; and;
eor     r0, r0, r2   a;  or;
     (a) ARM              (b) GA
```

Figure 1. `r0=(~r0&~r1)^r2` implemented in (a) ARM and (b) GreenArrays.

Implementing a superoptimizer for a new ISA is laborious. One must implement (i) a checker that verifies the equivalence of the target program and the source program; and (ii) a search strategy for finding a program that is optimal and correct on the test inputs. The equivalence checker is usually constructed using bounded verification, which requires translating programs into logical formulas. This effort requires debugging potentially complex logical formulas. Apart from the issues related to correctness and developers' effort, it is equally, if not more difficult to develop a search technique that scales to large programs. Our goal is to synthesize not merely a few instructions but multiple tens to hundreds of instructions. Therefore, the search strategy must be able to identify a correct program quickly in the huge space of all bounded-size programs expressible in the target ISA. As a result, we cannot simply enumerate all possible programs in the search space but need a more scalable approach.

In this paper, we present GREENTHUMB, an *extensible* framework for constructing *scalable* superoptimizers. Unlike existing superoptimizers, GREENTHUMB is designed to be easily extended to a new target ISA. Specifically, extending GREENTHUMB to a new ISA involves merely writing an ISA emulator. We illustrate GREENTHUMB's ability to support diverse ISAs by instantiating the framework to two very different ISAs: ARM [3] and GreenArrays 144 (GA) [12]. Figure 1 shows the output programs equivalent to program `r0=(~r0&~r1)^r2`, in ARM and GA. Note that GA is a stack-based architecture, so one instruction consists of an opcode without operands.

In addition to the retargetability of the framework, we also developed *hybrid-search technique* that helps GREENTHUMB scale to larger programs. Hybrid search employs multiple parallel search instances that collaborate in finding the optimal program. Each individual search instance implements either (i) *symbolic* search or (ii) *mutation-based* search. Symbolic search, based on symbolic program synthesis [13, 31], translates a specification given by the input program into a logical formula, and uses a constraint solver to find the optimal program. Mutation-based search [27], however, starts from an initial guess and uses mutation rules to search for the optimal program. Mutation-based search has been shown to handle larger programs compared to symbolic search, but it comes with the trade-off of losing the optimality guarantee. However, finding more optimal programs that might not be most optimal is better than finding nothing.

Hybrid search is effective because it allows for communication among different search instances that exchange information about the best program they have discovered so far. This communication enables hybrid search to exploit the best characteristics of both types of search techniques. Symbolic search has the ability to reason backward and make decisions using conflict clauses. Mutation-based search works with concrete candidate programs and uses a cost function to guide the search.

Apart from combining search techniques, we are also interested in improving the performance of each individual search technique. In this work, we focused on mutation-based search. We developed a new correctness cost function for mutation-based search that tries to capture how many mutations it would take to transform the current candidate program to a correct implementation of the given input program. In other words, the new correctness cost function captures *how correctable the candidate program is*, in contrast with prior correctness cost functions that only capture *how correct the output of the candidate program is*. The key insight behind the new cost function is to take into account values of intermediate variables and not solely those of the output variables.

Overall, GreenThumb is an enabling technology in the following three ways. First, it enables a *rapid* development of superoptimizers for different architectures. Second, it enables *reuse* of the same search technique across different architectures. Third, it enables development of *correct* search techniques; for example, in the new hybrid search, the formula generator (used by symbolic search) and the emulator (used by mutation-based search) are consistent with each other, because they are generated automatically from the same architecture description. In summary, this paper makes the following contributions:

- The design of a framework for constructing superoptimizers that is parametrized on the target architecture (§ 2 and § 3).
- A hybrid search technique (§ 5) that combines mutation-based and symbolic searches (§ 4).
- A new cost function for mutation-based search that considers intermediate values (§ 6).
- An evaluation of the new search technique and the new cost function on ARM, and GA architectures (§ 7 and § 8). Hybrid search was the only one returning optimal solutions for all GA benchmarks in all of its runs. The new cost function increased the number of runs that the superoptimizer found an optimal program by 20% on ARM benchmarks.

2. Framework Overview

GreenThumb is designed to be extensible to different architectures. In this section, we describe the framework from the point of view of the following three roles:

1. *User*: a person who wants to use GreenThumb to optimize code for a particular architecture.
2. *Builder*: a person who wants to extend GreenThumb to a new architecture.
3. *Researcher*: a person who wants to extend GreenThumb with a new search strategy.

Given a sequence of instructions P_g and live-out information L by a *user* and a target architecture, GreenThumb returns a more optimal sequence of instructions P_o that is equivalent to P_g with respect to L. Live-out information specifies the program locations that contain live values after executing the given instruction sequence. Such live-out information typically contains locations of the outputs and the values that the program should not modify.

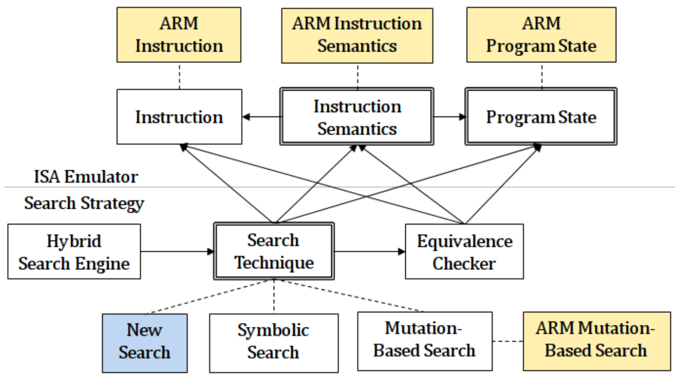


Figure 2. Framework Infrastructure.

To extend GreenThumb to a new ISA, *builder*s must write an emulator for the target ISA. Optionally, they can provide an architecture-specific performance model, which specifies the optimality criteria. Builders can then reuse existing search strategies implemented in GreenThumb, and they can additionally provide ISA-specific knowledge to help guide the search.

GreenThumb provides an interface by which a *researcher* can easily implement a new search strategy. The framework currently supports symbolic and mutation-based searches as the core techniques. We developed a new hybrid search, which runs multiple search instances of both symbolic and mutation-based search in parallel that communicate with each other. With this innovation, GreenThumb provides a convenient platform for evaluating different search strategies.

Framework Infrastructure Figure 2 depicts the overview of GreenThumb. The framework consists of ISA-emulator components and search-strategy components. We utilized inheritance to support retargetability of the framework.

We implemented each component as a class—shown as a box with single lines in Fig. 2—or an interface—shown as a box with double lines in Fig. 2. Solid edges represent dependency between components. For example, the solid edge from search technique to emulator components indicates that the search technique depends on those components. Dotted edges represent implement or extend relations. For example, the dotted edge from ARM instruction semantics to semantics indicates that ARM instruction semantics implements the semantics interface, and the dotted edge from ARM instruction to instruction indicates that ARM instruction extends instruction class.

The white components are provided by the framework, which includes the hybrid search engine. A *builder* constructs a superoptimizer for a new ISA by adding the yellow components to the framework, which extend some classes and implement the required interfaces. A *researcher* can experiment with new search techniques by adding more blue components, which implement search technique interface.

3. Extending GreenThumb to a New ISA

To extend GreenThumb to build a new superoptimizer, a builder must implement an emulator of the target ISA. Writing an ISA emulator in our framework involves three components: (i) describing the syntax of the instructions, which is done by defining an instruction class, (ii) defining a program state (§ 3.1), and (iii) defining the instruction semantics (§ 3.2). The builder then bridges the gap between the ISA emulator and search strategy by providing conversion functions (§ 3.3). Optionally, the builder can extend the instruction class to customize her own instruction representation and extend the mutation-based search to add new mutations (§ 3.4).

3.1 Program State

Program state is a user-defined data structure that represents the processor state. For example, a program state may include registers, stack memory, heap memory, and conditional flags. This data structure is used for representing input and output states corresponding to the processor state before and after program execution, respectively. It is also used for storing the liveness information of the values in the processor state. We allow the superoptimizer builder to define her own state representation because processor states of different architectures can vary drastically.

3.2 Instruction Semantics

The builder has to define the semantics of each instruction by writing an interpreter that, given an instruction and an input program state, returns the corresponding output program state. The emulator is built using Rosette, a solver-aided language [34] that is built on top of Racket.

The emulator can be used to interpret a sequence of instructions on a set of concrete inputs. This concrete emulation is required for mutation-based search (§ 4.2). Furthermore, because the emulator is written in Rosette, the builder obtains the following for free: (i) routines for proving the equivalence of two programs and finding a counterexample showing if the two programs are not equivalent, and (ii) an implementation of symbolic search (§ 4.1).

3.3 Interface Between ISA Emulator and Search Strategy

In order to make the functions in the search strategy components reason about different program states, we implement the functions in a way that can reason about any common state representation (CSR). A CSR S is defined as:

$$S := T \mid (S, S) \mid list\ of\ S \mid vector\ of\ S$$

$$T := \texttt{number} \mid \texttt{boolean} \mid \texttt{string}$$

The superoptimizer builder needs to bridge the gap between the custom program state and the CSR by implementing a (i) *serialize* function that converts a program state into a CSR, and (ii) *deserialize* function that converts a CSR into a program state. Because the equivalence-checking, correctness-cost calculation, and liveness-analysis functions are implemented to work with any CSR, different ISAs can reuse these common functions.

3.4 Instruction and Mutation-Based Search Extension

The framework provides a basic instruction structure that contains opcode and operands fields, and a mutation-based search that applies different mutation rules based on the default instruction structure. For instance, one mutation transforms only the operand in the instruction. To restrict the search to valid instructions, the search has to know the valid ranges of operands for the particular opcode in the instruction. Therefore, we require the builder to provide a function that returns the valid range of operands given an opcode. The framework also allows the builder to specify a set of legal mutations for a specific instruction. The superoptimizer builder can also extend the instruction structure to include more fields and extend the mutation-based search to introduce new mutation rules or disable existing mutation rules. This extensibility is required if one wants to implement an efficient and robust superoptimizer for different types of architectures, as described in § 7.

Additionally, GREENTHUMB provides a default performance model, which is the length of the instruction sequence. The builder can define a more accurate performance model that captures the specific property that the superoptimizer users want to optimize by extending the performance cost function of the emulator.

4. Existing Search Techniques

This section provides an overview of existing search techniques that we have implemented in GREENTHUMB. For each search procedure, we will first describe the general rules (or mechanisms) used and then describe the specific policies that govern how these rules are applied in practice.

We use $P, \ell \Vdash P_c, \mathcal{I}$ to denote the current state of a search instance. The specific goal of the search procedure is written to the left of $\Vdash$. In particular, the search procedure is trying to find a program that is equivalent to P and whose length is less than ℓ. To simplify the discussion, we assume that the search is trying to minimize the length of the program. In practice, the performance model might include other factors that model running time, power, etc. The current state of the search procedure is shown on the right of $\Vdash$. P_c is the current program that the search procedure is considering, and $\mathcal{I}$ is a set of inputs for P.

4.1 Symbolic Search

The symbolic search uses a constraint solver to find either a program P_c that satisfies the partial specification $\mathcal{I}$ or an input I for which the behavior of P and P_c differs. The SYMINIT, INDUCT, COUNTEREX, and DONE rules in Fig. 3 implement the counterexample guided inductive synthesis (CEGIS) approach [31].

Given P and ℓ, the SYMINIT rule initializes the search by starting with an invalid program ($\perp$) and an initial set of input-output behavior $\mathcal{I}$. The INDUCT rule synthesizes a program P_c' that satisfies the partial specification I, regardless of the current candidate program (P_c); for each $I \in \mathcal{I}$, $P_c'(I) = P(I)$. Furthermore, P_c' is of length less than ℓ. The COUNTEREX rule applies if there exists an input I such that $P_c(I) \neq P(I)$. This rule then adds I to the set $\mathcal{I}$. The DONE rule applies when $P_c \equiv P$ and $len(P_c) < \ell$. In other words, the DONE rule indicates that the search has successfully found a program that is equivalent to P and whose length is less than the given optimality criteria ℓ. Note that the DONE rule updates the optimality criteria from ℓ to $len(P_c)$.

The CEGIS process is performed iteratively until an optimal program is found. In our implementation, we use Rosette to generate the required queries to the solver. The efficiency of the symbolic approach, thus, depends on the efficiency of the solver, as well as having a good estimate of the value of ℓ; a smaller value of ℓ reduces the search space that the solver needs to explore. This observation motivated the way search instances communicate in our hybrid search (§ 5).

4.2 Mutation-based Search

We employ a stochastic superoptimization strategy [27] to implement the mutation-based search (MB). The mutation-based search is described by the INITRAND, INITCORRECT, COUNTEREX, and DONE rules in Fig. 3. Starting from a candidate program P_c, the mutation-based search applies rewrite rules (or mutations) to P_c to traverse the search space of programs. The search is guided by a cost function, $cost(P_c, P, \mathcal{I})$. In *synthesis* mode, $cost(P_c, P, \mathcal{I}) = eq(P_c, P, \mathcal{I})$. In *optimize* mode, $cost(P_c, P, \mathcal{I}) = eq(P_c, P, \mathcal{I}) + perf(P_c)$. $eq(P_c, P, \mathcal{I})$ indicates how close the behavior of P_c is to the given program P with respect to the set of inputs $\mathcal{I}$. The higher the value of $eq(P_c, P, \mathcal{I})$, the more the behavior of P_c differs from P. $perf(P_c)$ is the performance cost of P_c, which is equal to $len(P_c)$ in our simplified explanation.

The INITRAND rule initializes the mutation-based search using a random program P_r; we use MB$\langle$R$\rangle$ to denote such a mutation-based search. The INITCORRECT rule initializes the mutation-based search using a correct program P_c; we use MB$\langle$C$\rangle$ to denote such a mutation-based search. MB$\langle$R$\rangle$ and MB$\langle$C$\rangle$ run in synthesis and optimize mode, respectively. The MUTATE rule takes the search from the current program P_c to the program P_c'. P_c' is

$$\frac{}{P, \ell \Vdash \bot, \mathcal{I}}\ \textsc{SymInit} \qquad \frac{P, \ell \Vdash P_c, \mathcal{I} \quad \text{for each } I \in \mathcal{I}, P'_c(I) = P(I) \quad len(P'_c) < \ell}{P, \ell \Vdash P'_c, \mathcal{I}}\ \textsc{Induct}$$

$$\frac{P_r \text{ is a random program with length } \ell}{P, \ell \Vdash P_r, \mathcal{I}}\ \textsc{InitRand} \qquad \frac{}{P, \ell \Vdash P, \mathcal{I}}\ \textsc{InitCorrect}$$

$$\frac{P, \ell \Vdash P_c, \mathcal{I} \quad m \text{ is a mutation} \quad P'_c = mutate(P_c, m) \quad rand() < min\left(1, exp\left(-\beta \cdot \frac{cost(P'_c, P, \mathcal{I})}{cost(P_c, P, \mathcal{I})}\right)\right)}{P, \ell \Vdash P'_c, \mathcal{I}}\ \textsc{Mutate}$$

$$\frac{P, \ell \Vdash P_c, \mathcal{I} \quad P(I) \neq P_c(I)}{P, \ell \Vdash P_c, \mathcal{I} \cup \{I\}}\ \textsc{CounterEx} \qquad \frac{P, \ell \Vdash P_c, \mathcal{I} \quad P \equiv P_c \quad len(P_c) < \ell}{P, len(P_c) \Vdash P_c, \mathcal{I}}\ \textsc{Done}$$

Figure 3. Rules for symbolic and mutation-based search. Symbolic search employs SymInit, Induct, CounterEx, and Done rules. Mutation-based search employs InitRand, InitCorrect, CounterEx, and Done rules.

$$\frac{P, \ell_1 \Vdash P_{c1}, \mathcal{I}_1 \quad P, \ell_2 \Vdash P_{c2}, \mathcal{I}_2 \quad \ell_2 < \ell_1}{P, \ell_2 \Vdash P_{c1}, \mathcal{I}_1 \quad P, \ell_2 \Vdash P_{c2}, \mathcal{I}_2}\ \textsc{CommLen}$$

$$\frac{\begin{array}{c}P, \ell_1 \Vdash P_{c1}, \mathcal{I}_1 \\ P, \ell_2 \Vdash P_{c2}, \mathcal{I}_2 \quad P \equiv P_{c2} \quad len(P_{c2}) \leq \ell_1\end{array}}{P, \ell_1 \Vdash P_{c2}, \mathcal{I}_1 \quad P, \ell_1 \Vdash P_{c2}, \mathcal{I}_2}\ \textsc{CommSol}$$

Figure 4. Communication among different search instances.

generated from P_c using a mutation rule $mutate(P_c, m)$, where m represents a program mutation. The search accepts P'_c with the probability equal to the Metropolis ratio. If the search has found a program P_c with a cost zero with respect to $\mathcal{I}$, then the CounterEx rule searches for an input I on which P and P_c differ, and adds the input I to the set $\mathcal{I}$.

In the mutation-based search, a constraint solver is used to find the counterexample in CounterEx and to prove program equivalence in Done, similar to the case in symbolic search. However, the core search procedure involves computing the cost function and applying mutations to traverse the search space. Consequently, the efficiency of the mutation-based search is contingent on the cost function used to guide the search, and the nature of the mutations that are used in the rewrite rules. § 6 describes the cost function we developed, and § 7 describes the mutations we used. However, the mutation-based search can still get stuck at a local minima. In practice, we find that the mutation-based search finds an almost-optimal program quickly, but it is often unable to reach the optimal program. This observation motivated the way search instances communicate in our hybrid search (§ 5).

5. Hybrid Search

In this section, we describe the key insights behind our new hybrid search.

5.1 Communication between Search Instances

Figure 4 lists the two ways in which search instances communicate information. The current states of the two search instances are $P, \ell_1 \Vdash P_{c1}, \mathcal{I}_1$ and $P, \ell_2 \Vdash P_{c2}, \mathcal{I}_2$. In the CommLen rule, the first search instance updates its optimality criteria from ℓ_1 to ℓ_2. This occurs when the second search instance discovered an equivalent program of length ℓ_2. Merely communicating this new length ℓ_2 to the first search instance reduces the search space that the first search instance has to explore. In the CommSol rule, the second search instance communicates the actual program that it has found to the first search instance.

5.2 Symbolic and Mutation-Based: Better Together

In practice, the mutation-based search scales to larger programs compared to the symbolic search. For example, for ARM ISA, the symbolic search takes from 15 minutes to an hour to synthesize three instructions. In contrast, the mutation-based search takes less than a minute for the same example. This result would imply that given, for instance, a 32-core machine, we should run the mutation-based search on all 32 cores. However, we found that running a mix of mutation-based and symbolic search procedures is significantly better than running only instances of mutation-based search procedure.

Our *hybrid search* employs multiple parallel mutation-based and symbolic search instances. The search instances aid each other by exchanging information. MB⟨R⟩ instances reduce their search space by applying the CommLen rule. MB⟨C⟩ instances restart the search from a better program by applying the CommSol rule.

A mutation-based search instance can get stuck in a local minima. Often, a small sequence of mutations could be applied to this local minima to get to the optimal program. However, mutation-based search may reject such a series of mutations because the intermediate mutations may increase the cost of the program above the accepting threshold. When such a situation arises, the CommSol rule (Fig. 4) becomes extremely useful. Using the CommSol rule, a symbolic search instance receives this near-optimal solution, decomposes it into smaller programs, and optimizes the smaller programs individually. The symbolic search is considerably fast when synthesizing very short sequences of instructions. § 5.3 describes the two types of program decompositions, and § 5.4 describes the specific mix of symbolic and mutation-based search instances we use in practice.

5.3 Decomposition

Figure 5 lists the two types of decompositions that are applied during the search when the length of the program P is greater than a given threshold L. We use $[P_1, P_2, \ldots, P_k]$ to denote the concatenation of the program $P_1, P_2, \ldots$, and P_k.

The random-window decomposition, RandWindow, picks a random window P_2 of length less than L. In particular, P is randomly partitioned into P_1, P_2, and P_3, where the length of P_2 is less than L. It would be correct to optimize P_2 separately as in Chlorophyll [24]. However, optimizing P_2 in the context of the prefix program P_1 and postfix program P_3 leads to a more optimal program. For instance, the superoptimizer can make use of certain

$$\frac{P,\ell \Vdash P_c,\mathcal{I} \qquad len(P) > L \qquad [P_1,P_2,P_3] = P_c \qquad len(P_2) \le L \qquad [P_1,\underline{P_2},P_3],\ell \Vdash P_{c2},\mathcal{I}_1}{P,\ell \Vdash [P_1,P_{c2},P_3],\mathcal{I}}\ \textsc{RandWindow}$$

$$\frac{\begin{array}{c} P,\ell \Vdash P_c,\mathcal{I} \qquad len(P) > L \qquad [P_1,P_2,\ldots,P_k] = P \qquad len(P_i) \le L, 1 \le i \le k \\ [\underline{P_1},P_2,\ldots,P_k],\ell \Vdash (P_{c1},\mathcal{I}_1) \qquad [P_{c1},\underline{P_2},\ldots,P_k],\ell \Vdash P_{c2},\mathcal{I}_2 \qquad \ldots \qquad [P_{c1},P_{c2},\ldots,\underline{P_k}],\ell \Vdash P_{ck},\mathcal{I}_k \end{array}}{P,\ell \Vdash [P_{c1},P_{c2},\ldots,P_{ck}],\mathcal{I}}\ \textsc{SeqWindow}$$

Figure 5. The two types of program decompositions applied during search.

preconditions set up by the prefix P_1 when optimizing P_2. We use $[P_1,\underline{P_2},P_3],\ell \Vdash P_{c2},\mathcal{I}_1$ to denote the fact that we are optimizing P_2, which is underlined, in the context of P_1 and P_3.

The sequential-window decomposition, SEQWINDOW, partitions the program P into programs $P_1, P_2, \ldots P_k$, where the length of each P_i is less than L. $[\underline{P_1},P_2,\ldots,P_k],\ell \Vdash (P_{c1},I_1)$ denotes that P_1 is optimized to program P_{c1} in the context of the postfix $[P_2,\ldots,P_k]$. This optimized program P_{c1} is used as the prefix when optimizing P_2, as denoted by $[P_{c1},\underline{P_2},\ldots,P_k],\ell \Vdash P_{c2},\mathcal{I}_2$. In general, the optimized versions of the programs $P_1,P_2,\ldots,P_{i-1}$ are used as the prefix when optimizing program P_i where $2 \le i \le k$. In practice, we implement two variations of SEQWINDOW decomposition: *fixed window*, and *sliding window*, which was introduced by Chlorophyll [24].

The SEQWINDOW decomposition is more systematic, and might result in more optimal code. However, it could take a long time to get to optimizing the code towards the end of program P. In contrast, because RANDWINDOW decomposition optimizes a random subprogram, it is more likely to optimize parts of the code that occur towards the end of the program P.

5.4 Practical Configuration of Search Instances

The hybrid search consists of a single symbolic search instance using sliding-window decomposition, two symbolic search instances using random-window decomposition, three mutation-based MB⟨C⟩ instances, All remaining resources run mutation-based MB⟨R⟩ instances. Mutation-based search uses fixed-window decomposition. The decomposition threshold L of symbolic search is small, while that of mutation-based search is much larger.

We assign many threads to MB⟨R⟩ because mutation-based search is more sensitive to randomness than symbolic search. Thus, having more threads increases the chance of finding an optimal program. We do not need many MB⟨C⟩ instances, because there is a high chance that a search starting from a correct program will discover another correct program. The intuition is that some equivalent programs are far away from each other, but some are very close to each other in the search space, requiring fewer mutations to go from one to another.

6. A New Correctness Cost Function for Mutation-Based Search

Since the cost function is used for guiding mutation-based search, it is critical to the performance of search. Our new correctness cost function is a modification of the one used in the STOKE stochastic superoptimizer [27]. As in STOKE, we measure the correctness cost function of P_c with respect to the P using a set of inputs or test cases $\mathcal{I}$. Specifically, if $eq(P_c,P,I)$ is the correctness cost on one test input $I \in \mathcal{I}$, then the correctness cost of P_c with respect to P on a set of test inputs $\mathcal{I}$ is $\sum_{I \in \mathcal{I}} eq(P_c,P,I)$. In the rest of the section, we will focus on the definition of the $eq(\cdot)$ function.

6.1 Problems with Existing Correctness Cost Function

The correctness cost function $eq_{old}(P_c,P,I)$, used in STOKE, is defined as the number of non-matching bits between the live outputs of P_c and P, given I as the input. This correctness cost function captures how close the *output* of the candidate program P_c is to the output of the input program P; that is, it measures *how correct the output* of P_c is on the input I. However, eq_{old} does not capture how many mutations it would take to rewrite P_c to some correct implementation of P; that is, it does not measure *how correctable the program* P_c is on the input I.

Consider the following assembly programs in which `spec` is P that we want to optimize, and `r0` is the only live register. `f1` and `f2` are two candidate programs.

```
spec: // r0 = (~r0 & ~r1) ^ r2
      not r3, r0
      not r4, r1
      and r3, r3, r4
      xor r0, r3, r2

f1: // r0 = r2; r3 = ~(r0 | r1)
      or  r3, r0, r1
      not r3, r3
      mov r0, r2

f2: // r0 = r2
      mov r0, r2
```

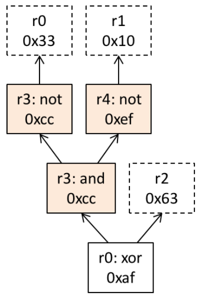
Assume 8-bit registers. Let the test input I' be {`r0=0x33`, `r1=0x10`, `r2=0x63`}. The output `r0` for `spec` is `0xaf`, while the output `r0` for both `f1` and `f2` is `0x63`. Consequently, $eq_{new}(\texttt{f1},\texttt{spec},I') = eq_{new}(\texttt{f2},\texttt{spec},I') = 4$. With this cost function, `f1` and `f2` have the same correctness cost.

The program `f1` implements $\neg(r0|r1) = (\neg r0\ \&\ \neg r1)$; Thus, inserting the instruction `xor r0, r3, r2` at the end of `f1` makes it equivalent to `spec`. In contrast, it would require many more mutations to tranform `f2` into a correct implementation of `spec`. Thus, the eq_{old} cost function fails to capture the fact that `f1` is more easily correctable compared to `f2`.

6.2 Proposed Correctness Cost Function

The new correctness cost, eq_{new}, is designed to capture how *correctable* the candidate program P_c is with respect to the input program P. In particular, `f1` can be corrected with fewer mutations compared to `f2`, and, hence, we define eq_{new} so that $eq_{new}(\texttt{f1},\texttt{spec},I') < eq_{new}(\texttt{f2},\texttt{spec},I')$.

The key insight behind our new cost function is to compare *intermediate values* computed by the P_c and P, and not just the output values. In particular, we say that P_c that produces similar intermediate values as that of P is *more correctable* than P_c that does not. Intuitively, the fact that P_c shares some common intermediate values implies that it is likely that some subprogram of P_c behaves in a similar fashion to some subprogram of P. Consequently, it is more likely that a fewer number of mutations are needed to transform P_c to a correct implementation of P.



(a) Computation DAG of `r0` in `spec`

test input =
`{r0=0x33, r1=0x10, r2=0x63}`

instruction	intermediate
`or r3, r0, r1`	0x33
`not r3, r3`	0xcc
`mov r0, r2`	0x63

intermediate value sets = {0x33, 0xcc, 0x63}

(b) Intermediate value set of `f1`

Figure 6. Example of computation tree and intermediate value set. (Left) Intermediate values are nodes with solid line. Input values are nodes with dotted line. Highlighted nodes are intermediate values that are covered by the program on the right.

Our new correctness cost function, $eq_{new}(\cdot)$ is defined as:

$$eq_{new}(P_c, P, I) = \frac{(1 + w(P_c, P, I))}{2} \times eq_{old}(P_c, P, I)$$

$w(P_c, P, I)$ is the fraction of the total number of intermediate values of P that are *not covered* by any intermediate value of P_c. We define an intermediate value as a result computed from an execution of an instruction. To compute $w(\cdot)$, we first obtain the computation tree of P on test input I. Figure 6 (a) shows the computation tree of `spec` on the example test input I'. We then compute a set of intermediate values produced by the candidate rewrite P_c. The set of intermediate values of `f1` is shown in Fig. 6 (b).

We define that an intermediate value v of P is *covered* if (1) v appears in the intermediate value set of P_c, or (2) v is reachable from any covered node v' in the computation tree. In the high level, if none of intermediate values of P_c matches intermediate values of P on test input I, $w(P_c, P, I) = 1$, and $eq_{new}(P_c, P, I) = eq_{old}(P_c, P, I)$. If intermediate values of P are all covered, $eq_{new}(P_c, P, I) = \frac{1}{2} eq_{old}(P_c, P, I)$. We designed the weighting function in this way to prevent the search from discovering the exact same program as the input program P.

The highlighted nodes in Fig. 6 (a) are the intermediate values of P that are covered. Nodes with value `0xcc` are covered because of condition (1), and the node with value `0xef` is covered because of condition (2). Therefore, $w($`f1`,`spec`,$I')$ = $1/4$, one intermediate value out of four is not covered. Note that the nodes with dotted lines are input values, and not intermediate values. As a result, the weighing function scales the correctness cost of `f1` down to 2.5, while the correctness cost of `f2` stays the same.

To enable the new cost function, currently the superoptimizer builder has to modify the emulator to return computation trees and an intermediate value set along with the normal output program state. However, the framework can be modified to construct trees and intermediate value sets automatically using the shadow execution concept by overloading necessary functions [23, 28, 29].

7. GreenThumb Instantiations

In this section, we describe instantiations of our GreenThumb superoptimizer framework for two very different architectures: ARM and GreenArrays. For each instantiation, we briefly describe the particular ISA, and list a few extensions to the mutation-based search that are specific to the ISA.

7.1 ARM

ARM is a RISC architecture that is widely used in many devices. We implemented a superoptimizer for ARMv7-A specifically, and modelled the performance cost function based on ARM Cortex-A9 [3]. The program state of ARM includes 32-bit registers, memory, and condition flags.

To handle optional condition code suffixes and shift operands found in ARM instructions, we added two new mutations to the mutation-based search. The first feature is the condition code suffix of an instruction, which indicates that the instructions will be executed if the condition is met. The second feature is the second *flexible* operand, which can either be a constant or a register with optional shift. One approach to handle these particular features would have been to treat an opcode with different condition-code suffixes and optional shifts as distinct opcodes. This approach would have resulted in a huge number of opcodes. Instead, we chose to extend the instruction class to include a condition-code field and an optional-shift field. Representing the instruction this way made it easier to implement the emulator. Because the instruction class was extended with additional fields, we extended the mutation-based search by adding two new mutation rules: (1) mutating condition code, and (2) mutating optional shift. The first mutation is only applicable for opcodes that have condition-code suffixes, and the second mutation is only applicable for opcodes with optional shift. Because the framework allows the builder to specify a set of legal mutations for a specific opcode, the mutation-based search only performs meaningful mutations.

We also implemented a superoptimizer for NEON, 128-bit SIMD architecture extension for the ARM. NEON has 64-bit registers, which are aliased with 128-bit registers. Currently, ARM and NEON superoptimizers run independently, but they could be combined to optimize ARM and NEON code together.

7.2 GreenArrays

The GreenArrays GA144 is a low-power processor, composed of many small identical 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 GA144 includes registers, stacks, memory, and a communication channel. This representation is similar to the one used in the superoptimizer implemented in Chlorophyll [24]. 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 contain exactly the same values.

Being a stack-base instruction set, most GA instructions only consist of an opcode with no operand. The only exception is fetch-immediate instruction that has an operand specifying the immediate constant value. We extended the mutation-based search to ensure that the search mutates an operand only for the fetch-immediate opcode.

During the development process, we noticed that sometimes the mutation-based search for GreenArrays finds a candidate program that is very close to being correct. Furthermore, the candidate program can be corrected by deleting an instruction in the middle of the program and inserting one more instruction at the end. However, the search can never find the correct program. To deal with this situation, we introduced a new type of mutation rule for GreenArrays, which we call *rotate mutation*, that picks a random instruction in the candidate program and moves it to the end. Using this new mutation rule, the search reaches the optimal solution more often.

The rotate mutation was not required for mutation-based search for ARM. Note that the effect of this new mutation rule can be simulated by the first replacing a random instruction by a `nop`, which effectively deletes that instruction, and then iteratively swapping the `nop` with an instruction that comes after as long as the semantics of the program is not affected. Because GreenArrays is a stack-based architecture, the `nop` can only be swapped with the next instruction. In contrast, in a register-based architecture such as ARM, the `nop` instruction could be swapped with an instruction further away. Consequently, it would take significantly fewer steps to simulate the rotate mutation for a register-based architecture compared to a stack-based architecture.

8. Experimental Evaluation

In this section, we present an experimental evaluation of the GREENTHUMB framework instantiated for the ARM and GreenArray architectures.

8.1 Comparing Search Strategies

The experiments in this section are designed to evaluate the effectiveness of (i) the hybrid search (§ 5) and (ii) the new correctness cost function (§ 6).

8.1.1 Methodology

We evaluate the following five versions of superoptimizers implementing different search strategies:

1. mutation-based starting from random programs (MB⟨R⟩)
2. mutation-based starting from the original program (MB⟨C⟩)
3. symbolic (S)
4. hybrid (H)
5. hybrid with old cost function (H')

In all versions, search instances communicate with each other as described in § 4. All superoptimizers, except H', use the new correctness cost function. S consists of two instances running sliding-window decomposition, and the rest running random-window decomposition. To evaluate the consistency of different search techniques, we ran each superoptimizer three times for each benchmark.

ARM Hacker's Delight Benchmarks consist of 16 of the 25 programs identified by [13] drawn from Hacker's Delight [35]. We excluded the first nine programs from our set of benchmarks because they are very small, and all five superoptimizers, except for symbolic, usually take less than 30 seconds to find the optimal programs. We used code produced by `gcc -O0 -march=armv7-a` as the input programs to the superoptimizers. The sizes of the input programs in this set range from 16 to 60 instructions. On this set of benchmarks, we ran all versions of superoptimizers using 32 search instances on a 16-core hyper-threaded machine. The timeout was set to one hour.

GA Benchmarks consist of frequently executed basic blocks from MD5 hash, SHA-256, FIR, sine, and cosine functions. We used code generated by Chlorophyll compiler before the superoptimization phase, and expert-written code as input programs to the superoptimizers. The sizes of the input programs in this benchmark suite range from 10 to 28 instructions. For these benchmarks, we executed 16 search instances on a 16-core Amazon EC2 machine. The timeout was set to 20 minutes.

8.1.2 Results

Figure 7 shows the performance costs of the best correct programs found in each of three runs of the two sets of benchmarks. The reported costs of each benchmarks are normalized by the cost of the known optimal program of that particular benchmark. An optimal program is a correct program with the lowest performance cost according to the defined performance model.

According to Fig. 7(b), on GA benchmarks, the hybrid superoptimizer was the only one that returned an optimal solution for all benchmarks in all of its runs. The second best was the symbolic superoptimizer, which found an optimal solution in at least one run for all benchmarks except for `sha2`.

ARM benchmarks are more difficult than GA benchmarks as seen in Fig. 7(a). None of the superoptimizers succeeded in finding optimal solutions for all benchmarks. However, the hybrid superoptimizer still performed the best; it failed to find optimal programs on only two benchmarks (i.e. `p23` and `p24`). In comparison, S and MB⟨C⟩ failed to find optimal programs on more than 5 benchmarks, and MB⟨R⟩ could not find any better program on 7 benchmarks.

The hybrid superoptimizer is not only better at finding optimal solutions but also faster at finding optimal solutions. Figure 8 reports the median time to find optimal solutions for the various superoptimizers. If a superoptimizer did not find an optimal solution on one or more runs on a benchmark, the table excludes that corresponding entry. It shows that most of the time, the hybrid superoptimizer was the fastest one to find an optimal solution.

The new cost function makes the hybrid superoptimizer more reliable and faster at finding optimal programs. Out of total 48 runs on ARM benchmarks, H found optimal solutions on 36 runs, while H' found optimal solutions on 30 runs; the new cost function increases the number of runs in which the superoptimizer found optimal solutions by 20%. By marking the best time between H and H' in Fig. 8, we can see that on the benchmarks on which H and H' consistently found optimal programs, H was faster than H' on seven ARM benchmarks, but H' was faster than H on only

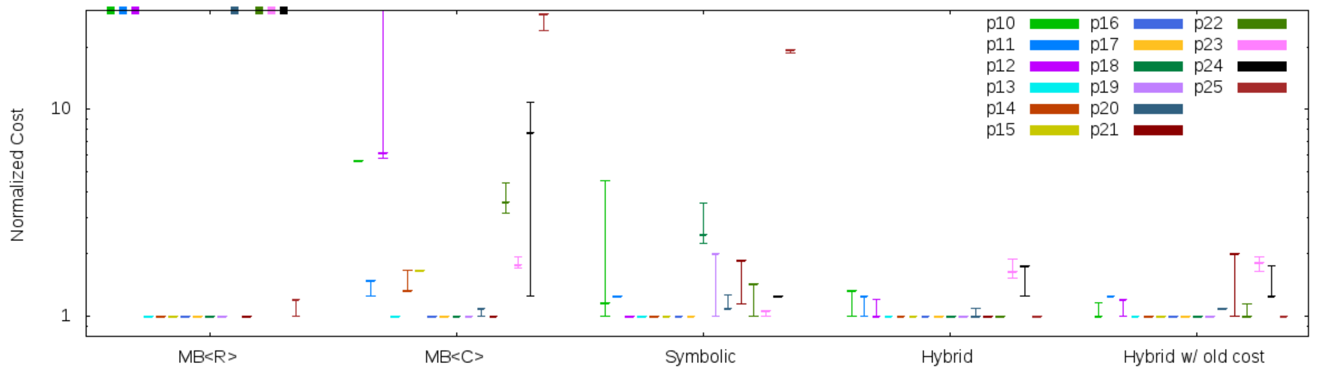
Benchmarks	MB⟨R⟩	MB⟨C⟩	S	H	H'
p10	-	-	-	-	-
p11	-	-	-	-	-
p12	-	-	**1941**	-	-
p13	**6**	**6**	141	7	7
p14	18	-	290	**15***	181
p15	31	-	171	**11***	154
p16	21	8	151	**7***	8
p17	-	334	176	**126***	270
p18	652	418	-	**376***	717
p19	1662	**85**	-	560	542*
p20	-	-	-	-	-
p21	2465	**122**	-	436*	-
p22	-	-	-	**2451***	-
p23	-	-	-	-	-
p24	-	-	-	-	-
p25	-	-	-	**229***	599

(a) ARM Hacker's Delight Benchmarks

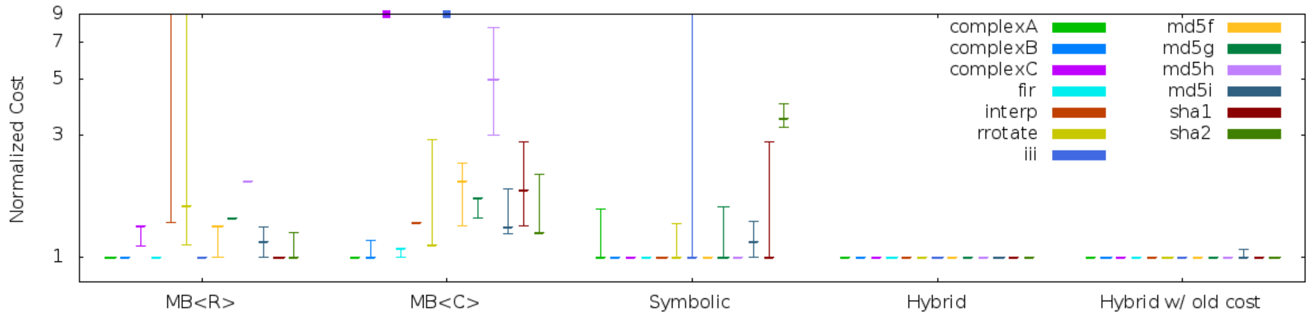
Benchmarks	MB⟨R⟩	MB⟨C⟩	S	H	H'
complexA	269	127	-	**48***	187
complexB	19	-	234	**17***	113
complexC	-	-	**10**	25	17*
fir	**18**	-	222	58*	131
interp	-	-	**109**	560	479*
rrotate	-	-	-	**118***	281
iii	616	-	-	**393***	679
md5f	-	-	439	**212**	67*
md5g	-	-	-	**852**	471*
md5h	-	-	**28**	38	35*
md5i	-	-	-	**492***	-
sha1	83	-	-	**38**	18*
sha2	-	-	-	**281***	604

(b) GA Benchmarks

Figure 8. Median time in seconds to reach optimal programs. **”-”** indicates that the superoptimizer version did not find an optimal solution in one or more runs. Numbers in **bold** denote the fastest time to find an optimal program between different search techniques, while * denotes the fastest time between different cost functions.



(a) ARM Hacker's Delight Benchmarks



(b) GA Benchmarks

Figure 7. Normalized performance costs of the best program found by the different superoptimizers. 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 of the chart. Timeout for ARM benchmarks was 60 minutes, and timeout of GA benchmarks was 20 minutes.

two ARM benchmarks. The new cost function provided 14-times speed up in the best case on `p15`, while it slowed down the search by 3-times in the worst case on `md5f`. On GA benchmarks, the cost function did not seem to help much, but it did not hurt the performance much either.

8.1.3 Detailed Explanations

Let take a closer look at why H is better than MB$\langle$R$\rangle$, MB$\langle$C$\rangle$, and S alone. Since H is a combination of MB$\langle$R$\rangle$, MB$\langle$C$\rangle$, and S, it has the strengths of both mutation-based and symbolic search. H is not just the best of MB$\langle$R$\rangle$, MB$\langle$C$\rangle$, and S. Because of the communication among different search instances, H can reliably find optimal solutions when none of the other techniques can.

Consider program `md5i`, which is one of the frequently executed routines in MD5 hash function. H was the only superoptimizer that consistently found an optimal solution in all runs. The success of H is due to the collaboration between different search strategies. The following lists the sequence of communications among search instances during a run of the hybrid search on `md5i`:

1. A symbolic search instance with sliding-window decomposition optimized the input program from 23 instructions to 15 instructions in half a minute.
2. An MB$\langle$R$\rangle$ instance learned from the symbolic search that an optimal solution likely consists of fewer than 15 instructions, so it reduced the search space by only searching for programs that have no more than 15 instructions. Four minutes later, this search instance found a better program with 11 instructions.
3. Finally, a symbolic instance with random-window decomposition received this new best program from the MB$\langle$R$\rangle$ instance, decomposed, optimized a portion of the program, and found an optimal program of length nine within three minutes.

On `p22`, a counting parity program, H was again the only superoptimizer that consistently found an optimal program in all runs. Within H, symbolic-search instances and an MB$\langle$C$\rangle$ instances alternatively found new best programs over time. The two-way communication between them led to finding optimal solutions reliably.

8.2 Improvement Over `gcc -O3`

In this section, we demonstrate that H, our hybrid search can discover faster programs in comparison to `gcc -O3` (called `gcc -O3` programs for abbreviation). We measure the execution time on ARM Cortex-A9. From the experiment in § 8.1, H found faster programs than `gcc -O3` on five benchmarks. On `p23`, H found programs that are slower than the `gcc -O3` program. For the rest of the benchmarks, H found programs that have similar performance as `gcc -O3` programs. Thus, we ran H again on `p23` program generated using `-O3`, and a faster program was found.

Additionally, H could find more optimal programs than `gcc -O3` programs on not only the Hacker Delight's programs, but also code inside libraries and kernels that are used in many applications.

Program	`gcc -O3` length	Output length	Search time (s)	Speedup
p11*	4	4	1729	1.22
p18*	7	4	383	2.11
p21*	6	5	436	1.81
p23	18	16	3	1.48
p24*	7	4	2200	2.75
p25*	11	1	229	17.76
wi-txrate5a	9	8	6	1.31
wi-txrate5b	8	7	136	1.29
mi-bitarray-1	10	6	1540	1.28
mi-bitarray-2	14	9	1453	1.05
mi-bitcnt-2-0	27	21	595	1.21
mi-susan-391	30	24	3225	1.15

Figure 9. Speedup over `gcc -O3` programs. * indicates programs that are optimized from `gcc -O0` taken the results from § 8.1.2. The rest are optimized from `gcc -O3`.

We compiled *WiBench* [38] (a kernel suite for benchmarking wireless systems) and *MiBench* [14] (an embedded benchmark suite) using `gcc -O3`. We then extracted basic blocks from the compiled assembly. We selected 13 basic blocks that contain more than seven instructions, contain only instructions supported by our superoptimizer, and have more data processing instructions than load and store instructions. For six out of these 13 ARM assembly programs, GREENTHUMB found faster programs compared to those generated by `gcc -O3`.

Figure 9 summarizes the characteristics of the faster programs found by GREENTHUMB and the speedup over `gcc -O3`. GREENTHUMB offers from 20% to 17-times speedup on some Hacker's Delight programs, and from 5% to 31% on some *WiBench* and *MiBench* code over `gcc -O3`.

8.3 Scaling to Large Programs

In this last experiment, we optimized very long sequences of instructions to evaluate the effectiveness of our program decomposition technique. We obtained `p23` and `p24` programs written in GA by using Chlorophyll compiler without superoptimization. The input `p23` program has 48 instructions, and `p24` program has 91 instructions. The length of both these programs is larger than the decomposition threshold L for GA mutation-based search, which is 40. The cost of the programs found by H with decomposition done by mutation-based instances is 22% and 10% lower than the cost of the programs found by H without the decomposition on `p23` and `p24`, respectively.

8.4 Developer's Effort

We use lines-of-code as a proxy to indicate the effort of building a superoptimizer. We compare two versions of a symbolic-search superoptimizer: one used in Chlorophyll [24] and one built with our framework. The former has 1,652 lines of Racket code; the latter has 579 lines of Racket code while being more advanced (with parallel execution feature).

9. Future Work

We plan to improve the efficiency of the mutation-based search by using symmetry-reduction techniques, which have been successfully used to speed up constraint solvers and model checkers [8, 22]. We plan to improve the efficiency of symbolic search by using a better encoding of synthesis queries to constraint formula that was described by [13].

We also plan to implement a new type of search strategy that uses offline exhaustive enumeration and online program stitching with equivalence class pruning. This particular technique has been shown to work extremely well for synthesizing functional programs in the ICFP competition [1]. The programs in the ICFP competitions take a single input and return a single output, which make program stitching very simple. However, we believe that this technique can be extended to programs with multiple inputs and outputs. Exhaustive enumeration alone has been used in many superoptimzers [4, 5, 11, 20, 35], but it does not scale by itself.

10. Related Work

We have already discussed most of the existing superoptimizers throughout the paper. Another well-known superoptimizer that we have not described elsewhere is Denali [16], which uses goal-directed search that employs the combination of enumeration for generating candidate rewrites and symbolic search to find the optimal program. All generated candidate programs are equivalent to the original program by construction since Denali enumerates programs by rewriting the original program according to algebraic identity. We do not include this approach as one of our search techniques because it is difficult to gather all possible algebraic identity rewrite rules when there are many complex instructions. As a result, the superoptimizer may miss an optimal solution.

There are several program synthesis tools besides Rosette that we could use for building our framework. First, Sketch [30] allows programmers to write complex programs while leaving fragments, called *holes*, of the code unspecified. We could implement the emulators using Sketch, and obtain symbolic search for free. Sketch does not provide an explicit functionality to do verification as does Rosette, but verification can be imitated by using its synthesis functionality. Second, Syntax-Guided Synthesis (SyGus) [2] introduces a standard program synthesis formula similar to SMT. SyGus currently supports three different synthesis engines: 1) enumerative, 2) mutation-based, and 3) symbolic. Two of the synthesis engines are the same as ours. We could have written a compiler that translates the implementation of an emulator to SyGus formula, and used the provided search techniques. However, SyGus search techniques currently do not support arrays or bitvectors that are larger 64 bits which are necessary for solving superoptimization problems. Apart from this limitation, if we built GREENTHUMB on top of SyGus, it would be harder for a superoptimizer builder to add new types of mutations. To do so, the builder would have to understand the back-end search engine and the SyGus AST, which is far different from normal representations of instructions.

Apart from the MCMC sampling technique used in our mutation-based search, there are many more search techniques that use mutations for finding an optimal solution with respect to a given cost function. The well-known classical techniques are simulated annealing, hill climbing, genetic algorithm, etc. In fact, genetic algorithms have already been used to automatically construct programs as known as genetic programming [18, 19, 32]. For example, PushGP is genetic programming for the Push programming language [32]. The correctness cost function used in PushGP is simply counting the number of incorrect outputs.

Our parallel hybrid search is similar to portfolio-based parallel SAT solvers such as Plingeling [6] ManySAT [15]. Understanding what techniques are effective in parallel SAT solvers may help us improve our hybrid search technique further.

11. Conclusion

We introduced GREENTHUMB, an extensible framework for building superoptimizers for diverse ISAs, implementing a variety of search techniques. Since only having an easy way to define an architecture would not be useful if we did not also have a scalable search technique, we developed a new hybrid-search technique that combines symbolic and mutation-based search. We showed that this hybrid search is better than either technique alone. Furthermore, GREENTHUMB also provides an interface to easily add new search techniques because we believe that with more search techniques, hybrid search might perform even better. In summary, we believe that GREENTHUMB represents an important step towards building usable, robust, and efficient superoptimizers.

References

[1] T. Akiba, K. Imajo, H. Iwami, Y. Iwata, T. Kataoka, N. Takahashi, M. Moskal, and N. Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time. Technical report, MSR, 2013.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*, 2013.

[3] ARM. *Cortex-A9: Technical Reference Manual*, 2012. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf`.

[4] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[5] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[6] A. Biere. Lingeling, plingeling and treengeling entering the sat competition 2013.

[7] J. Bungo. The use of compiler optimizations for embedded systems software. *Crossroads*, 15(1):8–15, Sept. 2008.

[8] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998.

[9] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins. picoarray technology: the tool's story. In *Design, Automation and Test in Europe*, 2005.

[10] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *Micro, IEEE*, Sept 2012.

[11] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *PLDI*, 1992.

[12] GreenArrays. *Product Brief: GreenArrays GA144*, 2010. URL `http://www.greenarraychips.com/home/documents/greg/PB001-100503-GA144-1-10.pdf`.

[13] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. Mudge, R. Brown, and T. Austin. Mibench: a free, commercially representative embedded benchmark suite. In *IEEE International Symposium on Workload Characterization*, 2001.

[15] Y. Hamadi and L. Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[16] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.

[17] M. Kandemir, N. Vijaykrishnan, and M. Irwin. Compiler optimizations for low power systems. In *Power Aware Computing*, Series in Computer Science. Springer US, 2002.

[18] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[19] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[20] H. Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[21] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4, Sept 2011.

[22] P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129, 2001.

[23] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[24] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[25] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency &#38; flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[26] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, July 2010.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[28] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[29] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[30] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis. EECS Department, University of California, Berkeley, 2008.

[31] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[32] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. In *Genetic Programming and Evolvable Machines*, pages 7–40, 2002.

[33] The Linley Group. Processor watch: Getting way out of box. `http://www.linleygroup.com/newsletters/newsletter_detail.php?num=5038`, 2013. Accessed: 2014-11-13.

[34] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[35] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[36] Wikipedia. List of arm microarchitectures. `http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures`, 2014. Accessed: 2014-11-13.

[37] C. Zhang. *Dynamically Reconfigurable Architectures for Real-time Baseband Processing*. PhD thesis, Lund University, 2014.

[38] Q. Zheng, Y. Chen, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge. Wibench: An open source kernel suite for benchmarking wireless systems. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013.