Acknowledgement

This work is supported in part by Qualcomm Innovation Fellowship, Microsoft Research Fellowship, Grants from National Science Foundation (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 Research.

GreenThumb: Superoptimizer Construction Framework

Phitchaya Mangpo Phothilimthana
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@qti.qualcomm.com

Abstract

Developing an optimizing compiler backend remains a laborious process, especially for nontraditional ISAs that have been appearing recently. Superoptimization sidesteps the need for many code transformations by searching for the most optimal instruction sequence semantically equivalent to the original code fragment. Even though superoptimization discovers the best machine-specific code optimizations, it has yet to become widely-used. We propose GreenThumb, an extensible framework that reduces the cost of constructing superoptimizers and provides a fast search algorithm that can be reused for any ISA, exploiting the unique strengths of enumerative, stochastic, and symbolic (SAT-solver-based) search algorithms. To extend GreenThumb to a new ISA, it is only necessary to implement an emulator for the ISA and provide some ISA-specific search utility functions. This paper demonstrates retargetability of GreenThumb by showing how to construct a superoptimizer for a small subset of LLVM IR.

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

Processors with new ISAs are constantly being developed [4, 6, 9, 10], and optimizing for them requires new architecture-specific optimizations. Peephole optimizations are introduced into compilers to perform such machine-specific optimizations by applying the rewrites specified by expert developers. Nevertheless, these human-written rewrite rules can miss many optimizations, and they can be buggy even in a well-developed compiler [7]. Additionally, some optimizations are difficult to realize because they are legal only under certain preconditions [17].

Superoptimization [8] is a method for obtaining a more optimal implementation of a given program fragment. Instead of applying predefined transformations, a superoptimizer *searches* for a sequence of instructions that is equivalent to a given reference program and optimal according to a given performance model. A few x86 superoptimizers [3, 5, 15] and a LLVM IR supertoptimizer [1] have been developed and shown to be very effective. For example, the stochastic superoptimizer offers 60% speed up over `gcc -O3` on a complex multiplication x86 kernel `gcc -O3` [15]. However, superoptimization has yet to become widely-used.

Superoptimization is not commonly used because implementing a superoptimizer for a new ISA is laborious, and the optimizing process can be slow. First, one must implement a search strategy for finding a candidate program that is optimal and correct on all test inputs, as well as a checker that verifies the equivalence of a candidate program and a reference program when the candidate program passes on all 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. Second, it is equally, if not more difficult to develop a search technique that scales to program fragments larger than ten or more instructions.

In this paper, we present GreenThumb, an extensible framework for constructing 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 describing the ISA—a program state representation, a functional ISA simulator, and a performance model—and some ISA-specific search utility functions. The framework provides a fast search strategy that can be reused for any ISA. GreenThumb is available on github at `https://github.com/mangpo/greenthumb`. The overview of the framework is presented in Section 2. In Section 3, we illustrate GreenThumb's ability to support diverse ISAs by showing some results from running the ARM and GreenArrays (GA) [6] superoptimizers built from GreenThumb. Last, we demonstrate how to construct a superoptimizer for a small subset of LLVM IR in Section 4.

2. Framework Overview

Figure 1 depicts the major components of GreenThumb and their interactions. At the core is the *cooperative* search algorithm that launches parallel search instances. Each instance consists of multiple components. First, the encoder-decoder parses a program into an IR. It is also used to print output optimized programs to files. On large programs, GreenThumb performs a *context-aware window decomposition* and uses a search technique to optimize a fragment p in the context of prefix p_{pre} and postfix p_{post}. The search technique searches for a candidate program that is semantically equivalent to the reference program but more optimal according to the given performance model. An ISA simulator evaluates the correctness of a candidate program on concrete test cases. If a candidate

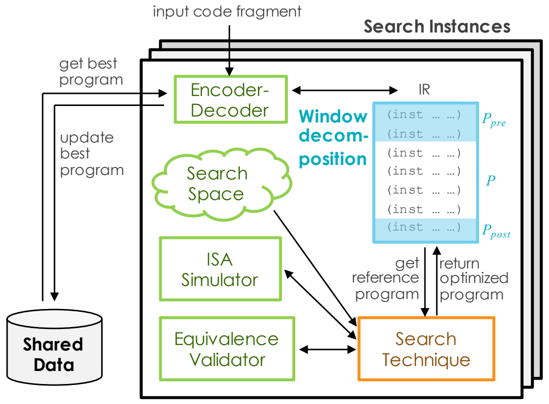

Figure 1. Overview of major components in GreenThumb

passes all test cases, the search technique verifies the candidate program against the reference program on all inputs using a constraint solver. If they are equivalent, and the candidate program is better than the current best program, the search instance updates the shared data. If they are not equivalent, the counterexample input is added to the set of concrete test cases.

Each search instance executes one of the three state-of-the-art search techniques provided in GREENTHUMB. Each instance is asked to find a program p such that $\forall(i,o) \in T \ . \ p(i) = o$, where T is the set of input-output test cases.

Symbolic search Our symbolic search exploits an SMT solver to perform the search. The search problem is written as a logical formula whose symbolic variables encode the choices of the program p. The formula embeds the ISA semantics and ensures that the program p computes an output o given an input i. Using Rosette [18], we obtain the symbolic search for free without having to implement a translator for converting programs to SMT formulas and vice versa. Compared to the other two algorithms, the symbolic search is slow, but it is able to synthesize arbitrary constants, which are sometimes needed in optimal code.

Enumerative search Our enumerative search implements the LENS algorithm [12], which refines *equivalence classes* only in the promising subspaces. It uses a goal-directed search strategy to exploring candidates forwards from inputs and backwards from outputs. The enumerative search synthesizes relatively small programs the fastest, but it does not scale to large programs by itself.

Stochastic search Our stochastic search explores the search space through a random walk using Metropolis Hastings acceptance probability with a cost function that reflects the correctness and performance of candidate programs [15, 16]. The stochastic search can synthesize larger programs compared to the symbolic and enumerative search because of the guidance of the cost function. However, it sometimes misses optimal programs because it can get stuck in local minima.

GREENTHUMB exploits the unique strengths of the three techniques to find a better program than each of them can alone. The search instances aid each other by sharing information about the best programs they have found so far (hence, the name *cooperative*). For example, the enumerative and symbolic search with the window decomposition may take the current best program and optimize some small fragments of the program further. The stochastic search may restart its random walk from the current best program found by the other. Additionally, when the symbolic search finds a better program with new constants, the other search techniques may include the new constants into its list of constants to try. The details about the cooperative search, context-aware window decomposition, and the LENS algorithm can be found in [12].

3. Case-Study ISAs

We used GREENTHUMB to build superoptimizers for ARM and GreenArrays (GA). Although ARM and GA are drastically different, our framework is able to support both ISAs. This demonstrates the retargetability of our framework.

ARM is a widely-used RISC architecture. An ARM program state includes 32-bit registers, memory, and condition flags. We extended GREENTHUMB for ARMv7-A and modeled the performance based on ARM Cortex-A9 [2]. In [12], we used the ARM superoptimizer built from GREENTHUMB to optimize some basic blocks generated from `gcc -O3` on Hacker's Delight benchmarks, WiBench (a kernel suite for benchmarking wireless systems), and MiBench (an embedded benchmark suite). Table 1 displays information about the basic blocks that the superoptimizer successfully optimized further. The 'runtime speedup' column reports the speedup measured by running the code on an actual ARM Cortex-A9 machine. The 'search techniques' column reports the search techniques used by the cooperative search that contributed to finding the final optimized code. According to the table, superoptimization offers significant speedup on many programs, and all the search techniques GREENTHUMB provides are necessary for finding the best programs.

Program	`gcc -O3` length	Output length	Search time (s)	Runtime speedup	Search techniques
hd-p18	7	4	9	2.11	E
hd-p21	6	5	1139	1.81	E, SM, ST
hd-p23	18	16	665	1.48	ST, E
hd-p24	7	4	151	2.75	ST, E
hd-p25	11	1	2	17.8	E
wi-txrate5a	9	8	32	1.31	SM, ST
wi-txrate5b	8	7	66	1.29	E
mi-getbit	10	6	612	1.82	SM, E
mi-bitshift	9	8	5	1.11	E
mi-bitcount	27	19	645	1.33	ST, E
mi-susan-391	30	21	32	1.26	ST

Table 1. Code length reduction, search time, runtime speedup over `gcc -O3` code, and search techniques involved in finding the solution. In the 'program' column, *hd*, *wi*, and *mi* represent code from hacker's delight, WiBench, and MiBench respectively. In the 'search techniques' column, SM, E, and ST represent the symbolic, enumerative, and stochastic search, respectively.

GreenArrays GA144 is a low-power, stack-based, 18-bit processor, composed of many small cores. Each core consists of two registers, two 8-entry stacks, and memory. Each core can communicate with its neighbors using blocking read and write instructions. A program state for GA, thus, includes registers, stacks, memory, and a communication channel, which is an ordered list of (data, neighbor port, read/write) tuples representing all the data a core reads and writes. We extended GREENTHUMB for GA and modeled the instruction timings according to [6]. We used the superoptimizer to optimize code generated from Chlorophyll [11] without any optimization. For MD5 hash, the largest program implemented in Chlorophyll, our superoptimizer found code that was 68% faster than the unoptimized code and only 19% slower than the expert-written code. In three critical functions of MD5, the superoptimizer actually found 1.3–2.5x faster code than the expert's.

4. Framework Demonstration

To demonstrate the retargeting of GREENTHUMB, our demo will implement a superoptimizer for a small subset of LLVM IR (32-bit arithmetic and logical instructions). GREENTHUMB is available on github at `https://github.com/mangpo/greenthumb`. The demonstration for LLVM IR is in the `llvm-demo` directory. In the demonstration, we highlight only the major steps of building a superoptimizer. In the rest of this section, the pronoun 'we' is used to denote the authors of the LLVM IR superoptimizer.

4.1 Organization

Figure 2 shows the relationship between the classes in the framework. GREENTHUMB is implemented in Racket and utilizes inheritance to provide retargetability. The classes in the top half of the figure will be used to describe the ISA, while the classes in the bottom half constitute the search procedure. To support LLVM IR, we will extend the classes in yellow as follows:

Step 1: Extend `machine%` to define the basic information about the ISA: the machine state and the supported instructions.

Step 2: Extend `parser%` and `printer%` to implement the encoder-decoder (see Figure 1).

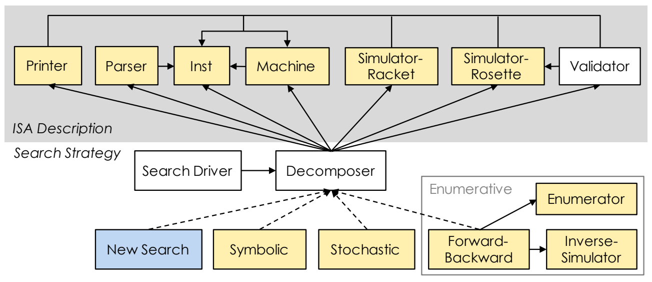

Figure 2. Dependency and class diagram. Each box represents a class. $X \rightarrow Y$ denotes X depends on Y, while $X \dashrightarrow Y$ denotes X is a subclass of Y. Users must extend the classes in yellow to create a new superoptimizer.

Step 3: Extend `simulator-racket%` and `simulator-rosette%` to implement the functional ISA simulators.

Step 4: Extend `symbolic%` to enable the symbolic search.

Step 5: Extend `stochastic%` to enable the stochastic search.

Step 6: Extend `forwardbackward%`, `enumerator%`, and `inverse%` to enable the enumerative search.

Step 7: Set up the cooperative search.

4.2 ISA Basic Description

We first define the basic description of the ISA, including the list of opcodes, the ISA bitwidth, and the structure of the program state.

Step 1: Machine We define such information in the class `llvm-machine%`, which extends the class `machine%`. First, we define how many bits are used to represent the smallest unit of value by setting the field `bitwidth`. Since we are supporting LLVM IR 32-bit arithmetic and logical instructions, we set `bitwidth` to 32. Next, we initialize the `opcodes` field to a vector containing all opcodes.

```
(define llvm-machine%
  (class machine%
    (super-new)
    (inherit-field bitwidth opcodes)
    (set! bitwidth 32)
    (set! opcodes '#(nop and or xor add sub and# or# xor#
         add# sub# _sub shl lshr ashr shl# lshr# ashr# _shl
         _lshr _ashr ctlz))))
```

Notice that LLVM has up to three instruction variants. For example, there are three subtractions: `sub`, `sub#`, and `_sub`. The prefix _ and suffix # indicate one of the input arguments is a constant. # indicates a variable as the first input argument and a constant as the second input argument; whereas, _ indicates the opposite.

Next, we implement the `set-config` and `get-state` methods to define the program state. GREENTHUMB restricts program states to contain only Racket primitive data structures: pair, list, and vector. Since this demonstration excludes load/store instructions, our program state is simply represented as a vector of variable values. The parameter describing the representation of the program state is, thus, an integer representing the number of variables. The method `set-config` takes in the program state parameter `config` and update the field(s) storing the parameter. This method is used during the initialization of a `machine%` object. `get-state` generates a fresh program state by initializing each value in the program state using the given lambda `init`. This method is used by the search techniques throughout the search procedure.

```
(define llvm-machine%
  ...
  ;; Field for storing program state parameter.
  (define vars #f)
  ;; Configure program state parameter.
  (define/override (set-config config) (set! vars config))
  ;; Generate program state using init lambda.
  (define/override (get-state init)
    (for/vector ([i (range vars)]) (init)))
```

4.3 Intermediate Representations

GREENTHUMB provides a default instruction representation, defined as (**struct** `inst` (`op args`)). In this demo, we will use this default representation. However, if necessary, users can extend `inst` with additional fields; for instance, the instruction representation for ARM has additional fields for conditional code and an optional shift argument. An instruction representation is a building block for constructing program representations. GREENTHUMB uses three levels of program representations:

Source is a plain-text source. For example, the program p below is an LLVM IR program in the source format:

```
%1 = lshr i32 %in, 3    ; %1   = %in >> 3
%out = shl i32 %1, 3    ; %out = %1 << 3
```

String IR is an IR after parsing a source, which is a vector of `inst`. Each `inst` includes an opcode in its field `op` and a vector of arguments in its field `args`. Opcodes and arguments are represented as strings. We assign an output variable to be the first element in an `arg` vector. The program p in the string-IR format is:

```
(vector
  (inst "lshr" (vector "%1" "%in" "3")))
  (inst "shl" (vector "%out" "%1" "3")))
```

Note that programs are stored using Racket data structures.

Encoded IR is an IR after encoding a String IR. It is also a vector of `inst`, but its `op` and `args` fields contain integer IDs instead of strings. An opcode ID is an integer indexing into `opcodes` in `machine%`. A variable ID is an integer that maps to a variable name. For constants, we simply convert strings to numbers. The program p in the encoded-IR format is:

```
(vector
  (inst 16 (vector 1 0 3))    ; 1 = %1 = 0 = %in
  (inst 15 (vector 2 1 3)))   ; 2 = %out
```

All components except `parser%` and `printer%` work with an encoded IR, because it enables representing programs with bitvector logic formulas used in the symbolic search and equivalence validator (which verifies the equivalence of two programs).

Step 2: Parser and Printer Since `parser%` and `printer%` are responsible for converting sources to encoded IRs and vice versa, we must extend them for LLVM IR by implementing:

- the class `llvm-parser%`, which parses LLVM IR source code into string-IR format.
- Three methods in the class `llvm-printer%`: `print-syntax-inst` prints string-IR program in source format; `encode-inst` converts string-IR to encoded-IR format; and `decode-inst` converts encoded-IR to string-IR format.

4.4 ISA Semantics

In order for GREENTHUMB to understand the semantics of LLVM IR and evaluate the performance of different code fragments, we have to implement an LLVM IR functional simulator and define its performance model.

Step 3: Simulator We must implement the methods `interpret` and `performance-cost` of the classes `llvm-simulator-racket%` and `llvm-simulator-rosette%`. The implementations of `llvm-simulator-racket%` and `llvm-simulator-rosette%` are in fact identical except that the former is implemented in **#lang** racket, while the latter in **#lang** s-exp rosette. The Racket simulator is used to interpret sequences of instructions on concrete inputs in the stochastic and enumerative search. The Rosette simulator is used by the symbolic search and equivalence validator. Although the Rosette simulator can also be used in the stochastic and enumerative search, it is slower than the Racket simulator.

We implement llvm-simulator-rosette% as follows. Our performance model is the length of the code fragment, excluding nop.

```
(define llvm-simulator-rosette%
  (class simulator-rosette%
    (super-new)
    (init-field machine)
    (define opcodes (get-field opcodes machine))

    (define-syntax-rule (bvop op)
      ;; finitize: truncate to 32 bits and convert to signed
      (lambda (x y) (finitize (op x y))))
    (define bvadd (bvop +))
    (define bvsub (bvop -))

    ;; Required method. Interpret a given program.
    (define/override (interpret program state)
      (define state-out (vector-copy state))
      ;; Interpret an instruction.
      (define (interpret-inst this-inst)
        (define op (inst-op this-inst))
        (define args (inst-args this-inst))
        ;; For one input variable and a constant.
        (define (xxi f)
          (define d (vector-ref args 0))
          (define a (vector-ref args 1))
          (define b (vector-ref args 2))
          (define val (f (vector-ref regs a) b))
          (vector-set! state-out d val))

        (define-syntax-rule (inst-eq name)
          (equal? name (vector-ref opcodes op)))
        (cond
         [(inst-eq `add#) (xxi bvadd)]
         [(inst-eq `sub#) (xxi bvsub)]
         ...))
        ;; Iterate and interpret each instruction.
        (for ([x program]) (interpret-inst x))
        ;; Return the output program state.
        state-out)

    ;; Required method. Evaluate performance cost.
    (define/override (performance-cost code)
      (vector-count
        (lambda (x) (not (= (inst-op x) nop-id))) code))))
```

Testing the Emulator

Now, we can interpret LLVM IR code!

```
> (define string-ir (send parser ir-from-string
     "%1 = lshr i32 %in, 3
      %out = shl nuw i32 %1, 3"))
> (define encoded-ir (send printer encode string-ir))
> (define input-state #(22 0 0)) ;; %in = 22
> (send simulator-rosette interpret encoded-ir input-state)
'#(22 2 16) ;; %in = 22, %1 = 2, %out = 16
```

4.5 Extending Search

Once we have the simulators working, we build a superoptimizer.

Step 4: Symbolic Search To enable the symbolic search, we need to implement the gen-sym-inst method to returns the *maximal* skeleton of an instruction, which can be constructed by using the provided lambda functions sym-op and sym-arg. The framework expects a program skeleton to be an inst with (sym-op) as its opcode and (sym-arg) as its arguments. The LLVM IR instructions that we support have up to three input/output arguments, so our *maximal* skeleton consists of three arguments.

```
(define llvm-symbolic%
  (class symbolic%
    (super-new)
    (inherit sym-op sym-arg)
    (define/override (gen-sym-inst)
      (inst (sym-op)
            (vector (sym-arg) (sym-arg) (sym-arg))))))
```

Defining Search Space for Other Search Techniques

Although we can use the same instruction skeleton from the symbolic search to define the search space for the other search techniques, it will not be very efficient. This is because the symbolic search exploits conflict clauses to prune the search space, while the other search techniques cannot. When the symbolic search tries an opcode, it can implicitly derive the valid values of arguments of the opcode. In contrast, the stochastic and enumerative search cannot derive this knowledge automatically.

Thus, we have to provide the following methods for computing this information in llvm-machine%. The method (get-arg-types opcode) should return a vector of types of opcode's arguments. An argument type indicates whether the argument is an input variable, output variable, or constant. (get-arg-ranges opcode live-in live-out) should return the valid values of opcode's arguments given live-in and live-out, specifying which variables are live before and after executing an instruction with the given opcode, respectively. live-in and live-out are given in the same format as a program state containing true (live), and false (not live). (update-live live-in this-inst) should return the live-out information after executing this-inst given a live-in information. Additionally, since the enumerative algorithm searches from both forward and backward directions, we have to implement the method (update-live-backward live-out this-inst), which returns the live-in information given a live-out information.

Step 5: Stochastic Search We need to implement the method (correctness-cost s1 s2 live) of the class stochastic%. [15] suggests a correctness cost to be the number of non-matching bits between the live outputs of program states s1 and s2 with rewards for correct (or nearly correct) values in wrong locations. stochastic% provides the method (correctness-cost-base s1 s2 live diff-bits) to calculate the suggested correctness cost when s1 and s2 are a vector of values, given a lambda function diff-bits that counts number of non-matching bits between two values. Thus, we can extend stochastic% as follows.

```
(define llvm-stochastic%
  (class stochastic%
    (super-new)
    (inherit pop-count32 correctness-cost-base)

    ;; Count non-matching bits between two 32-bit values.
    (define (diff-bits x y)
      (pop-count32 (bitwise-xor x y)))

    ;; Required method. Compute correctness cost.
    (define/override (correctness-cost s1 s2 live)
      (correctness-cost-base s1 s2 live diff-bits))))
```

Step 6: Enumerative Search The enumerative search consists of the classes forwardbackward%, enumerator%, and inverse%, as shown in Figure 2. forwardbackward% implements the LENS algorithm. It uses enumerator% to enumerates all possible instructions with all combinations of constants and variables that agree with the given live-in and live-out information. It explores the search space in a forward direction from input program states and a backward direction from output program states. The backward search relies on an inverse simulator provided by inverse%.

Thus, in llvm-enumerator%, we must implement the method (generate-inst live-in live-out) to return a generator [13] that generates all possible instructions. The `(gen-inverse-behavior this-inst)` method of `llvm-inverse%` will be invoked once for every instruction during the initialization. The method should generate and memorize the inverse behaviors of `this-inst` in the reduced-bitwidth (4-bit) domain to be used later during the search. One can simply generate the inverse behaviors of an instruction by executing the instruction on all possible program states in the 4-bit domain. The inverse behaviors can be stored as a map from an output program state to a list of input program states. Lastly, the method `(interpret-inst this-inst state)` in `llvm-inverse%` functions as an inverse simulator by looking up the memorized inverse behaviors of `this-inst` and returning a list of input states given an output state `state`.

Step 7: Window Decomposition and Cooperative Search Driver To help a search instance determine an appropriate size of a window in the context-aware window decomposition, we have to implement the method `(len-limit)` of `llvm-forwardbackward%` and `llvm-symbolic%` to return L, the size of code fragment (the number of instructions) that can be synthesized within one minute. According to our preliminary experiments, the size is 3 and 5 for the symbolic and enumerative search, respectively. The cooperative search varies the window size used for the different search instances; in particular, it uses L, $2L$, $3L$, and $4L$ window sizes.

We now have all the ingredients to run the complete cooperative search, which launches multiple search instances. The program to run the cooperative search is automatically generated by the setup script provided by the framework.

Testing the Search

We can use any search technique to quickly optimize:

```
%1 = lshr i32 %in, 3   ; logical shift right
%out = shl i32 %1, 3   ; shift left
```

which is taken from [14], to:

```
%out = and i32 %in, -8
```

We then try to optimize a larger code fragment. We use `clang` to generate the following LLVM IR code from a C program that rounds a number up to its power of two:

```
%1 = add i32 %in, -1
%2 = ashr i32 %1, 1    ; arithmetic shift right
%3 = or i32 %2, %1
%4 = ashr i32 %3, 2
%5 = or i32 %4, %3
%6 = ashr i32 %5, 4
%7 = or i32 %6, %5
%8 = ashr i32 %7, 8
%9 = or i32 %8, %7
%10 = ashr i32 %9, 16
%11 = or i32 %10, %9
%out = add i32 %11, 1
```

Running one search instance without the decomposition, the symbolic or stochastic search cannot optimize this program within five minutes, while the enumerative search can find a shorter program that uses count-leading-zeros instruction within six seconds:

```
%1 = sub i32 %in, 1
%2 = ctlz i32 %1       ; count leading zeros
%3 = lshr i32 -1, %2
%out = add i32 %3, 1
```

Using the context-aware window decomposition and four search instances, the symbolic search can also find the optimal or near-optimal solution within one minute, while the stochastic search seldom finds the solution. When we run our complete cooperative search, an instance that runs the enumerative search will find the solution quickly, and sometimes multiple instances of all search techniques can collaboratively find the solution more quickly than just the enumerative search alone.

5. Conclusion

We introduced GreenThumb, an extensible framework for constructing superoptimizers for diverse ISAs, providing a variety of search techniques. We believe that this framework represents an important step towards making superoptimization more widely used. That is, GreenThumb is an enabling technology supporting rapid development of superoptimizers and retargetability of efficient search techniques across different architectures. We plan to further explore the practicality of superoptimization in a compiler backend by targeting only frequently-executed parts of a program and using memoization.

Acknowledgments

This work is supported in part by Qualcomm Innovation Fellowship, Microsoft Research Fellowship, Grants from National Science Foundation (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 Research.

References

[1] Souper. URL `http://github.com/google/souper`.

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

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

[4] 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.

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

[6] GreenArrays. *G144A12 Chip Reference*, 2011. URL `http://www.greenarraychips.com/home/documents/greg/DB002-110705-G144A12.pdf`.

[7] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *PLDI*, 2015.

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

[9] 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*, 2011.

[10] Mill Computing, 2013. URL `http://millcomputing.com/`.

[11] 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.

[12] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *ASPLOS*, 2016.

[13] Racket. Generators, 2015. URL `http://docs.racket-lang.org/reference/Generators.html`.

[14] J. Regehr. Embedded in academia: A few synthesizing superoptimizer results. URL `http://blog.regehr.org/`.

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

[16] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, 2014.

[17] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Conditionally correct superoptimization. In *OOPSLA*, 2015.

[18] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541, 2014.