GREENTHUMB: Superoptimizer Construction Framework

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

Aditya Thakur
Google Inc.
Mountain View, USA
avt@google.com

Rastislav Bodik
University of Washington
Seattle, USA
bodik@cs.washington.edu

Dinakar Dhurjati
Qualcomm Research
Santa Clara, USA
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.

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, 7], 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 [5].

Superoptimization is a method for obtaining an 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 reference program and optimal according to a given performance model. A few x86 superoptimizers [2, 3, 11] and a LLVM IR supertoptimizer [1] have been developed and shown to be very effective. However, superoptimization has yet to become widely-used.

Superoptimization is not commonly used because implementing one for a new ISA is laborious, and the optimization process can be slow. First, one must implement a search strategy for finding a candidate program that is optimal and correct on the 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 instructions.

In this paper, we present GREENTHUMB, a framework for constructing superoptimizers, which is designed to be easily extended to a new target ISA, unlike existing superoptimizers. A longer version of this paper—which includes a detailed demonstration on how to build an LLVM-IR superoptimizer—can be found at [9]. GREENTHUMB is available at `https://github.com/mangpo/greenthumb`.

CC'16, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892233

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 an input, reference program into an IR. It is also used to print optimized programs to files. On large code fragments, GREENTHUMB performs a *context-aware window decomposition* and optimizes a fragment p in the context of prefix p_{pre} and postfix p_{post}. A search technique searches for a candidate program that is semantically equivalent to p in the context of p_{pre} and p_{post}, but better according to the given performance model. An ISA simulator is used to evaluate the correctness of a candidate program on concrete test cases. If a candidate passes all test cases, the search technique verifies the equivalence of the candidate program and the reference program on all possible inputs using an SMT solver. If they are equivalent, and the candidate program is better than the current best program, the new

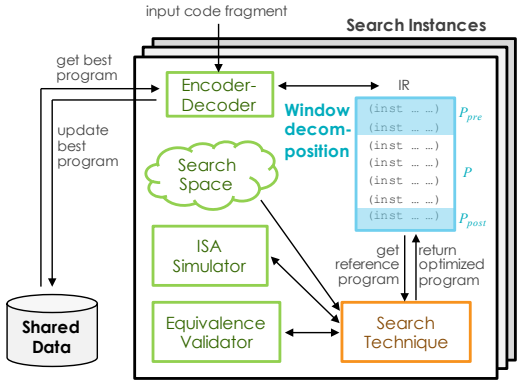

Figure 1. Overview of major components in GREENTHUMB

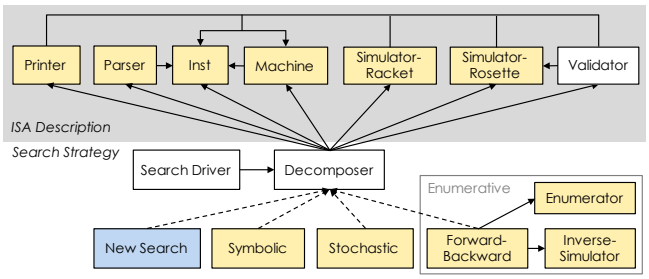

Figure 2. Dependency and class diagram. Each box represents a class. $X \rightarrow Y$ denotes X depends on Y, while $X \dashrightarrow Y$ denotes X is a subclass of Y. Users must extend the classes in yellow to create a new superoptimizer.

best program is saved to the shared data. If they are not equivalent, the counterexample is added to the set of concrete test cases.

Each search instance executes one of the three state-of-the-art search techniques. A *symbolic search* exploits an SMT solver to perform the search. An *enumerative search* implements the LENS algorithm [10], which refines the equivalence classes only in the promising candidate subspaces. A *stochastic search* explores the search space through a random walk with a cost function that reflects the correctness and performance of candidate programs [11]. The cooperative search 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 via the shared data. The details about the cooperative search, context-aware window decomposition, and three search techniques can be found in [10].

3. Extending for New ISAs

GREENTHUMB is implemented in Racket, and utilizes inheritance to provide retargetability. Figure 2 depicts the relationship between the classes in the framework, where yellow indicates the classes that need to be extended to support a new ISA.

The classes in the top half of the figure describe the ISA. Users extend these classes to implement a functional ISA simulator and a performance model for the ISA. A typical performance model is the sum of the instructions' average latencies of a given program.

The classes in the bottom half constitute the search procedure. Users extend the symbolic search by providing the *maximal* skeleton of an instruction (e.g., a skeleton of the instruction with the maximum number of arguments). Users extend the stochastic search by providing a function to calculate a correctness cost. Users extend the enumerative search by providing a generator to enumerate all possible instructions. Once the users have extended some search techniques for the ISA, they obtain the cooperative search for free. They can also adjust the window size used in the cooperative search's context-aware window decomposition.

4. Case-Study ISAs

We used GREENTHUMB to build superoptimizers for ARM and GreenArrays. Instantiating the framework for such drastically different ISAs demonstrates the retargetability of GREENTHUMB. More information about these two case studies can be found in [10].

ARM is a widely-used RISC architecture. We used the ARM superoptimizer to optimize basic blocks generated by `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 on an actual ARM Cortex-A9. 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 three search techniques 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 [4] 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. We used the superoptimizer to optimize code generated from Chlorophyll [8] 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 hash, the superoptimized code was actually 1.3–2.5x faster than the expert-written code.

References

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

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

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

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

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

[6] 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.

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

[8] 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.

[9] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Greenthumb: Superoptimizer construction framework. Technical Report UCB/EECS-2016-8, EECS Department, University of California, Berkeley, Feb 2016.

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

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