Synthesis for GreenArrays

CS294-80 Final Project

Phitchaya Mangpo Phothilimthana Tikhon Jelvis Rohin Shah

December 12, 2012

1 Problem Statement

This project applies program synthesis to generate efficient executable code on a low-power architecture made by GreenArrays. The architecture is dramatically different from other widely-used architectures, so the existing compiler optimizations do not work well. Additionally, because of its idiosyncratic architecture and set of instructions, it is extremely difficult to realize compiler optimizations for this chip. Even with other architectures, traditional deterministic compilers are not guaranteed to generate optimal code. In this project, instead of building a traditional compiler, we design and implement a synthesizer to generate the optimal machine code either in term of length or in term of the estimated running time from a given control-flow-free arrayForth program.

2 Background

GreenArrays 144 (GA144) is a stack-based architecture consisting of 144 independent asynchronous cores, with no clock and no shared memory. Each core contains a very small memory (under 300 bytes) and two short stacks. Because of this minimal selection of resources and its efficient core suspension protocol, GreenArrays consumes low amounts of energy. Compared to popularly used low-power microcontrollers such as MSP430, GreenArrays is 11 times faster and 9 times more energy efficient in a Finite-Impulse-Response application and 5 times more energy efficient per interrupt by an accelerometer, according to an experiment done by Nokia Research Lab in Summer 2012.

Each core of GreenArrays can only communicate with its neighbor ports by a blocking send-and-receive protocol. There is no message buffer between cores. As a result, the programmer needs to intersperse communication code into the computation code of a core and must be careful so as to prevent deadlocks and race conditions. Besides difficulty with implementing communication, another challenging task is partitioning a program into smaller parts (and allocating the parts to different cores) and handling large numbers.

Apart from the difficulty in partitioning and implement communication code correctly and efficiently, programming on a single core of GA144 is already extremely difficult because of its stack-based architecture, its limited set of instructions, and its small bit-width. The only programming language supported by GA144 is arrayForth, a dialect of Forth language. Its operators only work on the top elements of the stack as opposed to any arbitrary variables. In addition, arrayForth has a very limited set of instructions because it must be compiled into the machine code for GA144, called F18A, which only has 32 instructions. For example, there is no F18A instruction for inclusive or, multiplication, or division. Furthermore, since GreenArrays is an 18-bit architecture, it takes more work to perform 32-bit operations.

In this project, we focus on how to obtain efficient control-flow-free code for a single core of GA144.

3 Overview of our Approach

Our input is a correct, control-flow-free arrayForth program, which is the specification, and a sketch of an arrayForth program, in which the programmer leaves holes that can be filled by the synthesizer with any F18A instruction. Note that the set of F18A instructions (ignoring control-flow instructions) is a subset of arrayForth language. There are three important components - the formula generator, the verifier and the synthesizer. Given a (possibly incomplete) program, the formula generator generates an SMT formula that encodes the behavior of that program for any input state. The verifier checks whether a candidate program has the same input-output behavior as the specification, by asking an SMT solver whether there exists an input state for which the output of the two programs differ. The synthesizer takes a sketch and a set of input/output pairs, and asks an SMT solver to find some assignment of instructions to holes such that the resulting program when run on those inputs, produces the corresponding outputs, if such an assignment exists. We encapsulate the entire process in a CEGIS loop (described below). In order to optimize for time or length, we create another loop where we repeatedly ask the synthesizer for better and better solutions.

4 Synthesizer

4.1 Emulator and Compiler

In order to generate the input/output pairs, we have written an emulator for F18A machine code. The emulator faithfully copies the information given by GreenArrays. In particular, it also models the instruction fetch phase, and the program is stored in memory, so it supports self-modifying code. We have also written a compiler from arrayForth to F18A machine code, so that we can take naïvely written arrayForth, compile it to F18A machine code, and then optimize it.

4.2 Encoding

F18A consists of 32 instructions, but 24 of them are instructions that do not change the control flow of the program which are the instructions of our interest in this program. Thus, a 5-bit number is sufficient to represent an instruction.

The state of a program at any point consists of register a, register b, a 10-entry data stack, a 9-entry return stack, and a 64-word memory. In a default encoding, we represent registers by 18-bit bitvector and represent each stack and memory by one large bitvector. Based on past experience, representing an array of ints with one big bitvector makes the synthesizer run much faster than when representing it using an array of ints or an array of bitvectors.

Each instruction in a programs converts the old state into a new state. Therefore, we use static single assignment (SSA) form for the state of the program. We encode the formula of each instruction using a switch statement that alters the state of the program according to the value of the instruction as shown in Figure 1.

Each core can communication to its north, south, east, and west neighbors. Therefore, we use 4 bitvectors to represent sent data and 4 bitvectors to represent received data. The sent and received bitvectors are not in SSA form because they collect all sent and received data of the entire program.

With this encoding, we can use the same formula generator for both the specification and the sketch.

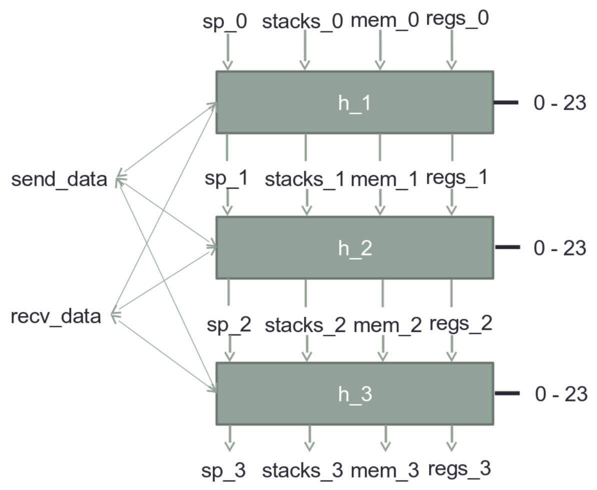


Figure 1: h_i represents an instruction at step i, whose value ranges from 0 to 23. h_i changes the state of the program from step $i-1$ to the next state i. *send_data* and *recv_data* represent communication to and from the 4 neighbors ports.

4.3 Specification and Sketch

The synthesis problem has two inputs: a specification and a sketch. The specification is a complete arrayForth program that is used to verify that the synthesized program is correct. The sketch is a partial arrayForth program with sections left out as "holes". The synthesizer then fills these holes with arrayForth instructions to create a program equivalent to the specification. If more information is put into a sketch, reducing the number of holes, the search space for the synthesizer is greatly reduced. This makes synthesis of larger programs tractable.

4.4 CEGIS

We used counterexample-guided inductive synthesis (CEGIS) to come up with valid programs. We start by getting a single input/output pair by running the specification on our emulator from a randomly generated start state. We then synthesize a program valid on that input. After this, we validate the program against the specification. If the program is valid, we have a solution. If the program is not valid, our verifier returns a new input/output pair with which we rerun the synthesis. We repeat this until we have a valid program, or until the synthesizer concludes that no such program exists.

Apart from modeling the semantics of the program, we also model its approximate runtime. This model is based on the cost of each instruction as provided by GreenArrays and is relatively simple because we do not support loops or branches. We use this model as an additional constraint on the programs we generate: the generated program has to be correct and must also run within some specified time. This additional constraint is used for finding optimal programs.

4.5 Different Constraints

By default, we generate programs that behave identically to the specification, including effects on the entire stack and memory. In practice, this is too restrictive. In many cases, we only care that the synthesized program have the same stack effect as the specification, so we support different correctness conditions. For example, we can ignore everything except the top of the data stack, or perhaps the top two elements. This allows greater freedom of optimization with a corresponding increase in synthesis time.

4.6 Optimizer

Our current synthesizer optimizes for the fastest program. We obtain the solution by iteratively calling the CEGIS loop for a faster program than the current solution (or the original program on the initial step) until it cannot find any better one. In the first version of our optimizer, we fix the code-length of the solution. However, most of the time the user provide more holes than necessary. Therefore, we call CEGIS loop with different length of code each time. Specifically, we binary search on the length. With this searching strategy, we can find the optimal program much faster.

Note that we do not support incremental solving; we currently do not take an advantage of restricting more and more constraints to get faster and faster solution. Hence, the binary search strategy results in faster synthesizing time. However, the fixed-length version might perform better if we support incremental solving.

4.7 Fake instructions

There are certain patterns of arrayForth instructions that are repeated very often. For example, there is no instruction for shifting by more than one bit; since this is a very common action, it appears in code fairly frequently. The synthesizer does not normally know about these sequences and would have to search through many different possibilities to find the right set of instructions for a shift. Moreover, the optimal shift code uses a loop which we cannot synthesize.

To deal with patterns like this, we added some fake instructions to the synthesizer. These instructions are essentially simple macros that expand to normal arrayForth. However, their semantics are encoded directly into the formula generation, which lets the synthesizer treat them as a single instruction. This simplifies the search space: the synthesizer does not have to re-derive these common patterns each time. This also lets us support a limited amount of loops: the code for shifting is implemented with a loop, but the semantics can be given to the synthesizer as if it was a normal instruction.

5 Optimizations

Each hole can be filled by multiple different instructions, making the search exponential in the number of holes. For actual programs, this means the search space is enormous. In order to make the synthesizer scale to harder problems, we added several optimizations.

Since the F18A machine code uses 18 bit numbers, we do as well. However, for many programs, if we can find a correct program for (say) 8 bit numbers, it will also work for 18 bit numbers. This means we can synthesize using fewer bits per number, significantly speeding up the process. However, in doing this, we may lose correctness. For example, trying to synthesize an inclusive or with one-bit numbers could result in the program +, which would not be a correct for 18 bit numbers. So, after synthesizing a program with a low number of bits, we verify it with 18-bit

program	orig len	opt len	orig time	opt time	len reduction	speedup	synth time
$x - (x\&y)$	8	2	15.5	3	4x	5.2x	20
$\neg(x - y)$	8	4	14	6	2x	2.3x	18
$x \mid y$	7	5	9	5	1.4x	1.8x	31
$(x + 7)\ \& - 8$	9	5	24	14	1.8x	1.7x	219
$(x\ \&\ m) \mid (y\ \&\ m)$	22	11	33	16.5	2x	2x	2852

Figure 2: Comparison on code length and runtime of original and synthesized programs including time to synthesize.

numbers to make sure that it is still correct. If not, we would have to synthesize again with a larger number of bits.

In the same vein, we do not need to model the core's whole memory. Most control-flow-free blocks of code do not need more than a few words of memory, so we can synthesize the program only modelling a few words making the model's state smaller and speeding up the process.

In some cases we know certain instructions will not be used. For example, if the specification does not access any memory, we can leave the memory-specific instructions out, shrinking the search space.

6 Experiments and Results

The first section shows the results of synthesizing programs from unoptimized code. The second section presents the synthesis times when applying different optimizations.

6.1 Synthesis Results

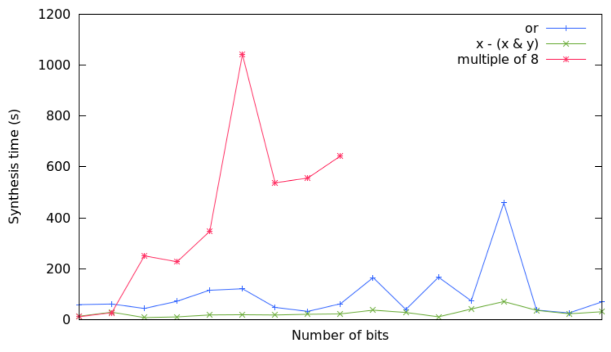
We took example code from *Hacker's Delight* by Henry S. Warren, Jr., translating the unoptimized examples to arrayForth to use as specifications. Figure 2 shows the code length and the estimated runtime of the spec and output programs as well as how long each example took to synthesize. We ran tests with different number of bits, pools of instructions and correctness constraints.

Our synthesizer takes between 15 seconds and 5 minutes to synthesize programs with about 8 unknown instructions, and under 50 minutes to synthesize a program with about 25 unknown instructions.

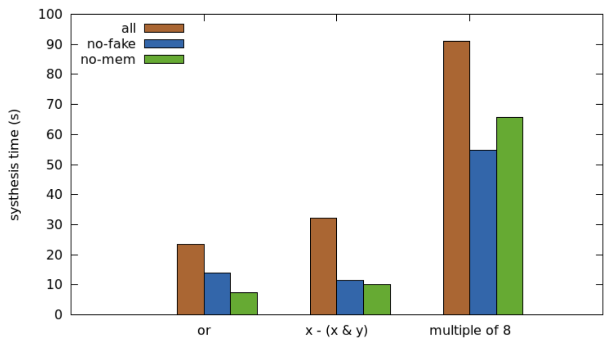
6.2 Optimization Results

We ran each benchmark on 3 different programs: $x|y$ (or), $x-(x\&y)$, and $(x+7)\&-8$ (multiple of 8). For each benchmark, we used default values for the other parameters: all instructions (without fake ones), all constraints, and 18-bit numbers.

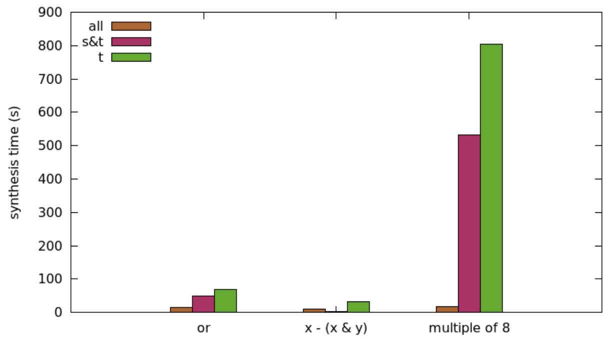
The first experiment compares different pools of instructions: all instructions including fake ones (all), instruction excluding fake ones (no-fake), and instruction excluding fake ones and memory-access instructions (no-mem). The second experiment compares different correctness constraints: everything (all), the top two elements of the stack (s&t), and the top of the stack (t). Figure 3 shows the synthesis time for the different varients. The last experiment compares using numbers of different bit widths.



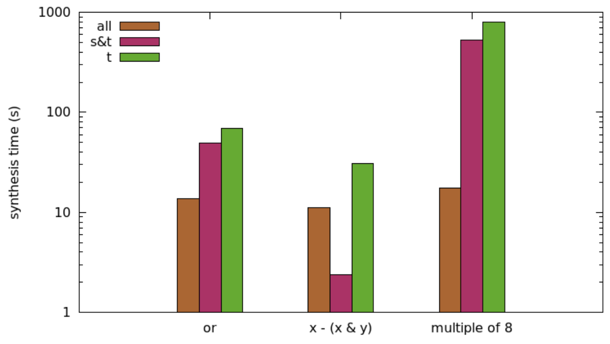
(a) Synthesis time when varying number of bits.



(b) Synthesis time when varying pools of instructions.



(c) Synthesis time when varying output state constraints.



(d) Synthesis time in log scale on y-axis when varying output state constraints.

Figure 3: Synthesis time when varying different parameters.

7 Future Work

In the future we would like to support loops, recursion, and conditionals. We can also support self-modifying code by modeling how instructions are stored in memory and fetched. However, such an encoding could have scalability issues.

In order to improve the scalability of the solver, we will experiment with different encodings of the program semntics as SMT formulas, especially since there has not been much research on encoding programs for stack-based architectures. We will also try using different solvers like KodKod and the Sketch backend which may be better tuned for our use-case.