SAT: A Solver Aided Toolchain for building accurate CPU emulators from scratch

Andrew L. Chronister
University of Washington
chronal@cs.washington.edu

Bill Zorn
University of Washington
billzorn@cs.washington.edu

Abstract

Accurate emulation of a CPU, with or without documentation at hand, can be a difficult endeavor. Even when a full manual is available, processor behavior may be left undefined for some combinations of inputs or sequences of operations. For completely accurate emulation, it may be desired that these undocumented behaviors be faithfully executed by the emulator the same way the original hardware does. We present SAT, a Solver Aided Toolchain for creating emulators from hardware observations. The central premise of SAT is that if we have a microprocessor available, we can use its observable behavior to create an emulator which is accurate by construction, rather than rely on manuals and documentation. SAT accomplishes this goal for simple microprocessors such as the TI MSP430 (used in their Launchpad platform) by using a debug interface to collect data, which is then used with previous work in software synthesis to create machine-specific plugins for a basic extensible emulator framework.

1 Introduction

Program synthesis traditionally operates by starting with a high-level specification of a program's intended behavior, and then automatically producing programs which satisfy that behavior. Often, the context of program synthesis may also demand the most optimal programs that satisfy a particular specification. With SAT, we leverage previous work in program synthesis with the observed behavior of instructions on a physical microprocessor as the specification, and the program length as a cost function, to generate an emulator which is correct by construction over all possible inputs and has minimal implementations for each instruction. This allows for accurate emulation of a processor in software without requiring an arduous process of trial and error by a human programmer. This also allows for the creation of documents which specify processor behavior in all cases, which may be desired if official documentation leaves certain behavior undefined.

2 The Pipeline

We decompose the toolchain into three distinct stages: measurement, synthesis, and emulation. This allows us allocate resources more effectively for each part, for example by running long-running measurement on one machine connected to arrays of target devices, and CPU-intensive parallel synthesis on a different machine. Each stage could be implemented in completely distinct environments, but we opted to use Racket as the implementation environment for consistency and re-usability. Within Racket, we use Rosette[3] as a platform for symbolic queries and as a provider of bit-vector arithmetic features (which, conveniently, can thus also be symbolic).

2.1 Measurement

The pipeline begins with a measurement stage. The microprocessor's debug interface is used to program a harness that executes instructions with known inputs and then collect the outputs at a known memory location. This harness is also responsible for clearing the registers to zero, and catching all possible interrupts that could be thrown as a result of invalid instruction execution. The precise shape of this harness is processor-specific, but must comprehensively record all state changes resulting from execution of the instruction(s) of interest.

In the case of the MSP430, it was sufficient for us to record the outputs for the possible combinations of the status register flags at the beginning of the operation, and known input registers. For operations with a small input space, such as 8 bit operations, it's feasible to collect data for all possible combinations of these inputs. For operations with a larger input space, such as 16 bit and wider

operations, we use dense random sampling to get a representative portion of inputs that is likely to be sufficient for synthesis.

This data is accumulated into a series of input/output tables. The contents of these tables will depend on the breadth of possible results from instruction execution – for example, if an instruction can result in interrupts being thrown, then this must be able to be reflected in the i/o table – but pragmatically should be as minimal as possible to save on disk space and later processing time. For simple operations, the direct inputs, output register, and status register are sufficient to describe the behavior of the instruction, so we simply store these as a sequence of `(input . output)` pairs, one per line.

2.2 Synthesis

Once we have data about the microprocessor's behavior over a sufficiently large number of inputs, we use it to generate assertions that can be fed into a metasketch synthesis suite to find a program which yields correct results for all known inputs. In this pipeline, we use Synapse[1] to generate sequences of bitvector operations that satisfy the input/output assertions. For efficiency reasons, we use a relatively low number of samples from our measured input/output data to generate assertions initially (as the SMT solver Synapse uses performs better with fewer assertions), and then check the resulting programs against the entire set of data so that we can add samples that provide counterexamples if necessary. This approach is similar to existing methods of counterexample-guided synthesis (CEGIS)[2].

In addition to the recorded microprocessor observations, the synthesis must also be parameterized by the bit-width at which the operation takes place (in order to find minimal programs that don't involve unnecessary masking or shifting), the synthesis strategy to use, and the expected maximum program length for the instruction.

Synthesis strategies are ways of iterating over the input values to produce the output value. Most operations can simply operate on the entire input value at once, for instance `add` can simply perform a simple bitvector addition over the two inputs. However some operations can only be succinctly expressed as a series of smaller operations that operate on part of the input at a time. For example, a binary-coded-decimal (BCD) `add` may need to be expressed as an operation that applies to four consecutive bits at a time. The strategy to use for a given operation is human-specified in our implementation, but it would be feasible for the synthesis stage to iterate over the known strategies if its initial attempt is unsatisfiable in the specified program length.

A maximum program length is used to reduce the potential running time if the strategy hint for an instruction is incorrect. The running time for program synthesis increases exponentially with the program length and the number of operations available to the synthesizer, so it is undesirable to allow synthesis to run indefinitely if it is clear from circumstantial observation that the processor's behavior should be relatively straightforward to specify.

We synthesize both the programs to calculate the direct results of CPU operations and the programs to calculate the resulting status-register flags by decomposing the status register from the data set into its component bits and running synthesis on each one separately.

2.3 Emulation and Co-simulation

The emulator consists of the modelled processor state and the evaluation process that manipulates that state. Processor state can be idealized as a list of memory maps whose interface is a map from addresses to values in memory. These can in practice be more specialized data structures depending on the purpose of each memory map. For example, the MSP430 can be modelled as a two such maps: a vector of register-size bitvectors for the register file, and an interval-map of word-size bitvectors for the memory as seen by the processor (which, on actual MSP430 devices, is an abstraction of RAM, FRAM, I/O devices, and so on).

The evaluation process consists of decoding instructions, reading from memory maps, executing instruction behavior, writing back to memory maps, and then advancing the instruction stream. Of these parts: decoding must be hand-written; reading from memory maps is partially abstract enough to be implemented by the framework and partially hand-written, though it could be synthesized as a future direction for the emulator; evaluating instructions is entirely synthesized; writing back to the memory maps is partially abstract enough to be implemented by the framework and partially hand-written, with the same qualifier as for reading; and finally, advancing the instruction stream is handled by the emulator.

The bitvector programs from the synthesis stage of our pipeline can be used almost directly as Racket code in the emulator, with some modification to reformat them out of SSA form and replace references to flag bits with actual extraction of the bits from the status register. Knowing the opcode of each instruction and the name corresponding to that opcode allows us to generate the dispatcher, so that the synthesized program can simply be included into the implementation-specific module.

The correctness of the complete emulator can be verified by running programs simultaneously on both the emulator and on the target hardware. The emulator framework provides co-simulation capability, so that the same

code can be loaded on both the hardware and the emulator, and then single-stepped or run until halting. Registers and memory can be examined in both the hardware and the emulator to check for inconsistencies.

3 Future Directions

The SAT pipeline at present is equipped to measure and synthesize operations with two inputs. The logical development of the measurement and synthesis pieces of this pipeline would be to add support for operations with only one input, which includes control flow operations. Additionally, future work on SAT could add inference of addressing mode encoding and calculation using the domain-specific language already defined in the emulator. This would reduce the amount of hand-written code needed to add support for a new processor using the pipeline. Also, emulation of interrupts is necessary for a comprehensive processor emulator, and synthesizing the particular behavior of these would be a necessary step towards completeness.

More generally, in order to make this work useful to the programming community at large, it would be necessary to ensure other processors can be fully and efficiently emulated by the system. It is unclear whether CISC processors that may have complicated internal dispatch for some instructions would be possible to emulate using this system, which would limit its usefulness in some areas.

References

[1] JAMES BORNHOLT, EMINA TORLAK, D. G., AND CEZE, L. Optimizing synthesis with metasketches. *POPL* (2016).

[2] SOLAR-LEZAMA, A. Program sketching. *International Journal on Software Tools for Technology Transfer 15*, 5-6 (2013), 475–495.

[3] TORLAK, E., AND BODIK, R. A lightweight symbolic virtual machine for solver-aided host languages. *PLDI* (2014).