Constrained Conservative State Symbolic Co-analysis for Ultra-low-power Embedded Systems

Shashank Hegde†, Subhash Sethumurugan†, Hari Cherupalli*, Henry Duwe‡, and John Sartori†
† University of Minnesota, *Synopsys Inc., and ‡ Iowa State University
{hegde031,sethu018,jsartori}@umn.edu,haricherupalli@gmail.com,duwe@iastate.edu

ABSTRACT

Symbolic simulation and symbolic execution techniques have long been used for verifying designs and testing software. Recently, using symbolic hardware-software co-analysis to characterize unused hardware resources across all possible executions of an application running on a processor has been leveraged to enable application-specific analysis and optimization techniques. Like other symbolic simulation techniques, symbolic hardware-software co-analysis does not scale well to complex applications, due to an explosion in the number of execution paths that must be analyzed to characterize all possible executions of an application. To overcome this issue, prior work proposed a scalable approach by maintaining conservative states of the system at previously-visited locations in the application. However, this approach can be too pessimistic in determining the exercisable subset of resources of a hardware design. In this paper, we propose a technique for performing symbolic co-analysis of an application on a processor's netlist by identifying, propagating, and imposing constraints from the software level onto the gate-level simulation. This produces a more precise, less pessimistic estimate of the gates that an application can exercise when executing on a processor, while guaranteeing coverage of all possible gates that the application can exercise. This also reduces the simulation time of the analysis, significantly, by eliminating the need to explore many simulation paths in the application. Compared to the state-of-art analysis based on conservative states, our constrained approach reduces the number of gates identified as exercisable by up to 34.98%, 11.52% on average, and analysis runtime by up to 84.61%, 43.83% on average.

KEYWORDS

symbolic execution, symbolic simulation, gate-level analysis, hardware/software co-analysis, constraints

ACM Reference Format:
Shashank Hegde†, Subhash Sethumurugan†, Hari Cherupalli*, Henry Duwe‡, and John Sartori† . 2021. Constrained Conservative State Symbolic Co-analysis for Ultra-low-power Embedded Systems. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3394885.3431157

ASPDAC'21, Tokyo Odaiba Waterfront, Japan,

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431157

1 INTRODUCTION

A large number of emerging computing applications have ultra-low-power requirements [15, 17]. Notable among these are the internet of things, sensor networks, wearable electronics, and biomedical devices. The ultra-low-power requirements of these applications are due to the fact that the applications are either energy-constrained (e.g., battery-powered applications), power-constrained (e.g., energy-harvesting applications), or both. The embedded systems used by these applications typically consist of a low-power microcontroller / microprocessor running a single application over and over throughout its operational lifetime.

Based on the application-specific nature of many emerging ultra-low-power systems, a recent line of work has proposed application-specific power and energy reduction techniques that identify hardware resources (e.g., gates) in a processor that cannot be exercised by the application running on the processor and eliminate the power used to support those resources [7, 9, 10]. However, such application-specific optimizations can only be safely applied if an analysis technique can guarantee that the application running on the processor will never use the resources for any possible execution of the application, for any inputs. Eliminating gates or power for resources that could be used by the application could lead to incorrect execution of the application. For example, power gating a gate that was incorrectly identified as "unused" but is actually exercised by an application can result in the application producing incorrect outputs or crashing. Given the need for guarantees and the inability to achieve such guarantees through input-based application profiling, recently-proposed application-specific power management techniques rely on a symbolic simulation [4] of the application on the processor hardware to identify hardware resources that are guaranteed to not be used across all possible executions of an application. By propagating symbols that represent unknown logic values for all inputs to an application, it is possible to determine all possible hardware resources that could be used by the application in an input-independent fashion [9, 10]. Recent work has demonstrated that the input-independent activity profiles generated by such a symbolic simulation of an application running on a processor can be leveraged to identify worst-case timing, power, and energy characteristics for a low-power system and to eliminate power used by resources that the system's captive application is guaranteed to never use [7, 9, 10].

Symbolic hardware-software co-analysis is also related to symbolic execution [5]; however, it differs in that co-analysis considers every possible execution of a given application for all possible inputs, rather than determining a specific set of inputs that will test the application. Also, prior works on symbolic execution do not consider the application-specific effects of software on hardware.

Symbolic hardware-software co-analysis suffers from the same scalability limitations characteristic of symbolic simulation and execution – namely, the state-space explodes for applications with

complex control structures, due to a large number of possible execution paths. Many heuristics have been proposed to tame the scalability problems of symbolic simulation and execution [2, 3, 5, 6, 16]; however, existing heuristics for scalable symbolic simulation and symbolic execution cannot be applied to the symbolic co-analysis used for application-specific power management, since existing heuristics sacrifice perfect coverage to improve scalability.

This problem was handled in prior work [8] by maintaining *conservative* states at PC-changing instructions. A conservative state is formed by merging two states and replacing the differing bits in the states with Xs, representing unknown logic values. A conservative state encompasses a superset of all observed states every time the simulation re-visits the PC. If a state is a sub-state of the conservative state maintained at the PC, that state has already been simulated, and execution from the state can be terminated.

However, this conservative over-approximation still treats the application as a black box, and hence, suffers from the pessimism of marking too many gates as exercisable, potentially leaving significant benefits on the table. This is due to the nature of conservative state construction, where states are merged by replacing locations that are different with Xs; thus, the number of states represented by the resulting super-state can be exponentially more than the number of states used to generate the conservative state (e.g., see Section 2). This can lead to covering states that are not possible in the original application.

In this paper, we propose a constrained conservative state symbolic hardware-software co-analysis technique that characterizes the behavior of an application by analyzing the binary to determine constraints, e.g., bounds of a particular memory element. Such bounds can be used to *constrain* the value of the memory element from being overly pessimistic (i.e., containing too many Xs), leading to fewer gates marked as exercisable and reduced simulation times. Our technique is based on the observation that simulating from conservative states can mark many gates as exercisable that the application cannot actually exercise, and that conservative state-based simulation can unnecessarily explore execution paths that are not actually possible for the application.

This paper makes the following contributions.

- We show that the over-approximation of prior scalable symbolic hardware-software co-analysis is significant, due to treating the application as a black box, and demonstrate that applying higher-level constraints from the software can significantly mitigate this over-approximation.
- We propose a *constrained conservative state symbolic hardware-software co-analysis* technique that translates constraints on variables at the software level to constraints on memory elements in the processor-memory system, enabling reduced pessimism over state-of-the-art techniques in the number of gates that are marked as exercisable by up to 34.98%, and by 11.52% on average.
- We show that in addition to reducing the number of gates identified as exercisable for an application, our co-analysis technique also reduces analysis time. Compared to existing techniques, analysis time is reduced by up to 84.61%, and by 43.83% on average.

2 BACKGROUND AND MOTIVATION

Co-analysis techniques presented in prior work [7, 9, 10] identify all exercisable gates for an application in a processor through symbolic simulation of the application on the processor netlist. The symbolic simulation analysis technique is described in the **black** text of Algorithm 1. Unfortunately, this co-analysis technique cannot analyze applications with complex control flow or infinite loops. To resolve this issue, prior work [10] proposes maintaining conservative states for each PC-changing instruction (e.g., conditional branch). A conservative state is a state that covers all simulated states observed at a particular PC-changing instruction. An execution path is simulated only if the current state is not a subset of a previously observed conservative state, in which case a more conservative state is created by merging the current state with the conservative state maintained for the PC-changing instruction and continuing simulation from the new conservative state. This algorithm is described by the **black** and **blue** text in Algorithm 1. The conservative approximation technique described above enables a scalable gate activity analysis that completes in a small number of passes through the application.

Figure 1c illustrates how conservative state is generated and how maintaining conservative states can significantly improve simulation time. The example code in Figure 1b (compiled from C-code in Figure 1a) represents a simple subroutine that updates an internal variable (represented by `r13` (`p`)), based on an external value (represented by `r14` (`q`)), over 16 iterations (tracked by `r5` (`i`)). The first section of the code (red) initializes the registers `r5`, `r13`, and `r14`. The next two sections (blue and yellow) are the loop body, where `r13` is compared against `r14`. If `r14` $\geq$ `r13`, line 7 is executed to increase `r13` by `r14`. Otherwise, simulation iterates again, after decreasing the loop counter (`r5`) in the next section (green). After exiting the loop, we `return` from this subroutine.

Figure 1c shows the execution tree and the values of two registers `r13` and `r5` at various states that the processor reaches during execution. The simulation starts in the red block and reaches the end of the blue block. Since `r14` contains Xs, the subsequent jump `jnc`'s path is inconclusive, and an X propagates to the PC. We split the simulation to execute both branch paths – the yellow block and the green block. The state of the processor at the end of the blue block is represented as S_0, and the states of the processor at the start of false and true paths are represented as S_0^F and S_0^T, respectively. The same convention is used for the rest of the states in the tree. Each state in the table contains two rows for the values of the registers `r13` and `r5`. The upper row represents the value of the register observed when the simulation reaches the corresponding point in the execution tree. The lower row represents the conservative value computed by merging this value with the previous conservative state observed at this point. Simulation proceeds using this conservative value instead of the observed value. One example of conservative approximation is that of register `r5` for state S_1. Since S_1 and S_0 correspond to the same PC, we build a conservative state to represent both the states S_1 and S_0 when we simulate down S_1; this is achieved by replacing the values that differ between the two states with Xs. In the case of `r5`, the two states differ in the least significant 5 bits, which are replaced by Xs to represent both the states. This *X-ification* of the states leads to skipping execution of several states downstream and thus a faster completion of application analysis.

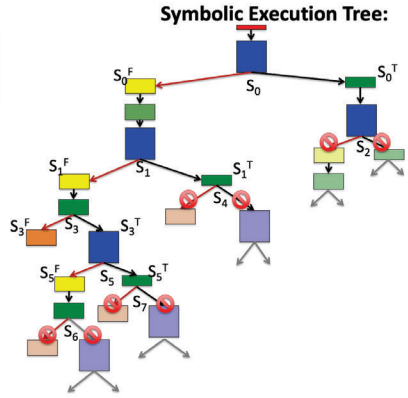
However, the conservative over-approximation of `r5` at S_1 represents not only the two states merged but also all 32 states representable by varying the lower 5 bits of `r5`. Therefore, when we execute the instruction `dec r5` in the green block just before state S_3, the value 16'bXXXXX can represent 32 different values, including 16'b0 and 16'b1. Decrementing 16'b0 by 1 results

```
int p=2, q=*val;int i;
for(i=16; i>0; i--){
      ...
      if (p < q){
          p += q;
      }
} // i > 0; i--
return;
```

(a) Example C Program.

```
1. mov #16, r5
2. mov #2, r13
3. mov &200, r14
loop:
4.    ...
5. cmp r13, r14
6. jnc then
7. add r14, r13
then:
8.dec r5
9.jnz loop
10. ret
```

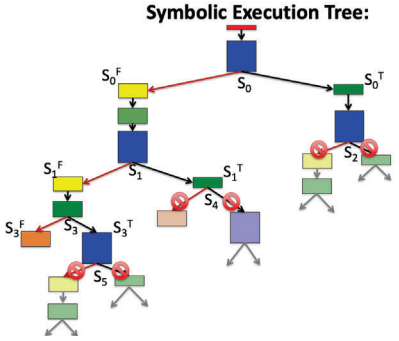
(b) Compiled MSP430 Program.



Conservative States:

State	PC	r13	r5
		R13(c_state)	R5(c_state)
S_0	6	0b0000000000000010	0b0000000000010000
		0b0000000000000010	0b0000000000010000
S_1	6	0bXXXXXXXXXXXXXXXX	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000xxxxx**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000xxxxx**
S_3	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_4	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_5	6	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_6	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_7	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX

(c) Conservative state-based scalable symbolic co-analysis can analyze applications with infinite loops and input-dependent branches by simulating conservative states that capture the activity of multiple possible states.



Constrained Conservative States(CCS):

State	PC	r13	r5
		R13(cc_state)	R5(cc_state)
S_0		0b0000000000000010	0b0000000000010000
	6	0b0000000000000010	0b0000000000010000
S_1	6	0bXXXXXXXXXXXXXXXX	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000xxxxx**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000xxxxx**
S_3	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0b00000000000xxxxx
S_4	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0b00000000000xxxxx
S_5	6	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0b00000000000xxxxx

1 <= r5 < 17

(d) Constraining memory elements based on bounds from the software level reduces pessimism in estimating the number of gates marked as exercisable and also reduces the number of paths that need to be explored.

Figure 1: Illustration of Symbolic Simulation with Conservative Approach vs Constrained Conservative Approach

in 16'b1111111111111111 (two's complement arithmetic), while decrementing 16'b1 by 1 results in 16'b0. To represent both these states, r5's value becomes 16'bXXXXXXXXXXXXXXXX. Unfortunately, this represents all the 2^{16} possible values for a 16-bit number, leading to exercising many more gates in the processor than necessary, since r5 only actually assumes values between 0 and 16. Since $0 \leq$ r5 < 17, we can constrain the value of r5 to 16'b000000000000XXXXX, as shown in Figure 1d. This not only reduces the number of gates toggled; it also reduces the number of execution paths simulated, leading to faster convergence.

Based on this motivation, we develop an approach to encode information about constraints on variables from the binary, propagate them during co-analysis, and enforce them on the variables to reduce pessimism in conservative state analysis. We present our approach in the next section.

3 CONSTRAINED CONSERVATIVE STATE (CCS) SYMBOLIC CO-ANALYSIS

In this section, we explain Constrained Conservative State Symbolic Co-Analysis. Our co-analysis tool (see Figure 2) is based on the observation that certain constraints on variables at the software level are lost when the application is simulated at the gate-level, leading to overly pessimistic estimates of the hardware resources

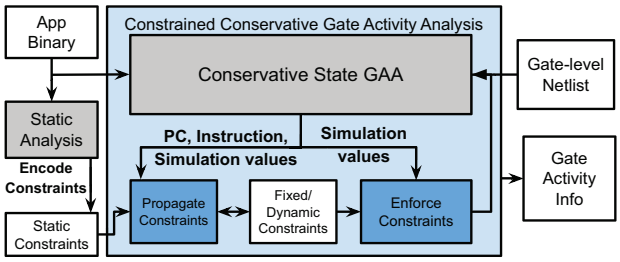


Figure 2: Methodology for CCS

(i.e., gates) needed to execute the application. We translate software-level constraints to the gate level in three steps. First, we encode high-level program constraints as constraints on the operand values of static instructions. Our tool generates these constraints from a pattern-based static analysis of the application binary. Second, these encoded constraints are loaded into the conservative symbolic simulator and propagated from source operands to destination operands during simulation. Third, when operands containing Xs are updated by an instruction, encoded and propagated constraints are applied so that the operands' symbolic values observe the constraints. Pseudocode of our implementation is shown in Algorithm 1 and Algorithm 2. Changes to the conservative symbolic co-analysis

Algorithm 1 Constrained Conservative State Symbolic Co-analysis

```
1.  Procedure GateActivityAnalysis(app_binary, design_netlist)
2.  Initialize all memory cells and all gates in design_netlist to X
3.  Load app_binary into program memory
4.  Propagate reset toggle signal
5.  s ← State at start of app_binary
6.  Symbolic Execution Tree T.set_root(s)
7.  Unprocessed execution points queue, U.push(s)
8.  C.init() // Initialize conservative system state map
9.  C_T.load_constraints() // Load Static constraints map
10. while U != ∅ do
11.     e ← U.pop()
12.     if e.isConditinalBranch() and e.PC ∈ C then
13.         a ← C.getState(e.PC)
14.         if e.isConservativeSubstateOf(a) then
15.             continue
16.         else
17.             e ← buildConservativeState(a, e)
18.             C ← C.update(e.PC, e)
19.         end if
20.     else if e.isConditionalBranch() then
21.         C ← C.add(e.PC, e)
22.     end if
23.     while e.nextPC != X and !e.END do
24.         e.setInputsX() // set all peripheral port inputs to Xs
25.         e' ← propagateGateValues(e) // perform simulation for this cycle
26.         if e'.aboutToCommit() then
27.             // instruction will be committed in the next cycle
28.             c_t ← getConstraints(C_T, e'.PC)
29.             e' ← propagateConstraints(e', c_t) // transfer constraints, source to destination
30.             e' ← enforceConstraints(e', c_t)
31.         end if
32.         e.annotateGateActivity(e, e') // annotate tree point with activity
33.         e.addNextState(e') // add to execution tree
34.         e ← e' // process next cycle
35.     end while
36.     if e.nextPC == X then
37.         for all a ∈ possibleNextPCVals(e) do
38.             e' ← e.updateNextPC(a)
39.             U.push(e')
40.             T.insert(e')
41.         end for
42.     end if
43. end while
```

in Algorithm 1 are presented in **red**. In the following subsections, we explain each step in greater detail.

3.1 Encoding Constraints from Binary

In order to constrain simulation values, our tool must know what memory element's values should be constrained, what its valid set of values are, and at what execution points those constraints are valid. As shown in Figure 3, our tool takes value bound constraints (e.g., 0 to 17) on instruction operands (e.g., r5) for specific instructions (e.g., the `mov`, `dec`, and `jnz` instructions at PCs 3, 9, and 10, respectively).

An example instruction pattern is shown in Figure 3. Many possible static analyses at different abstraction levels, from C compiler to binary analysis, could be used to generate constraints, with varying trade-offs of coverage and precision [13, 14, 19]. The exploration of these trade-offs is beyond the scope of this paper. For our work, we chose to use a pattern-based binary analysis approach where we map known binary patterns resulting from high-level program structures (e.g., loops and if statements) into constraints (e.g., register holding a loop iterator is bounded between its initialization and termination values at loop boundaries). We have identified nine such patterns involving different types of loops and nested loops. Note that for pattern-based analysis, the relevant patterns can depend on compiler options. Our library of patterns covers the most common patterns observed in our benchmark set (see Section 4).

3.2 Propagating Constraints

Once we encode all the constraints, we load them into the co-analysis tool as `Fixed` (i.e., immutable) constraints on operands

Algorithm 2 Constraint Enforcement

```
1.  // e : Execution state of the processor
2.  // c_t : Constraint
3.  Procedure enforceConstraints(e, c_t)
4.  if e.isOutputOutOfBounds(c_t) then
5.      if e.isMemoryOp() then
6.          e ← handleMemoryEnforcement(e, c_t)
7.      else
8.          e.dstRegVal ← genConstrainedVal(e.dstRegVal, c_t)
9.      end if
10. end if
11. return e

12. //e : Execution state of the processor
13. // c_t : Constraint
14. Procedure handleMemoryEnforcement(e, c_t)
15. if containsX(e.memAddress) then
16.     for all addr ∈ possibleAddresses(e.memAddress) do
17.         if isAddressInBounds(addr, c_t.addressConstraint) then
18.             if e.memOperation == read then
19.                 val ← generateConstrainedConservativeVal(val,
                        e.dMemory[addr], c_t.valConstraint)
20.             else if e.memOperation == write then
21.                 e.dMemory[addr] ← generateConservativeVal(e.val, e.dMemory[addr])
22.             end if
23.         end if
24.     end for
25.     if e.memOperation == read then
26.         e.dataBus.put(val)
27.     end if
28. else
29.     addr ← e.memAddress
30.     if e.memOperation == read then
31.         val ← e.dMemory[addr]
32.         e.dataBus.put(val)
33.     else if e.memOperation == write then
34.         e.dMemory[addr] ← e.val
35.     end if
36. end if
37. return e
```

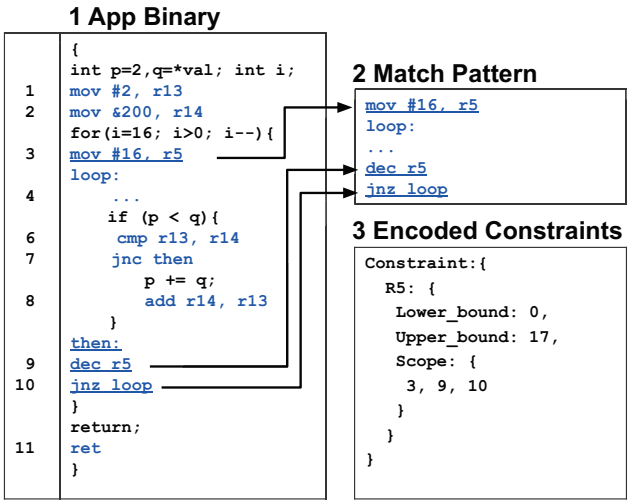


Figure 3: Example of constraint encoding during static analysis of the application binary.

(i.e., register and memory values) at specific static instructions, and we start symbolic co-analysis. During co-analysis, we intercept every instruction when it is about to be committed in the processor pipeline, read the constraints on the instruction's source operands, and update the constraint on the destination operand if that operand does not have a `Fixed` constraint at the current PC. This updating creates a `Dynamic` constraint for the memory element.

Consider the instruction `mov #2, r13`, with `r13` having no constraint before the instruction is executed. At the end of the execution of the instruction, we will have a constraint on `r13` as $2 \leq$ `r13` < 3, representing its constant value.

Consider another instruction, `add r5, r13`, with constraints on `r5` as $1 \leq$ `r5` < 17 and on `r13` as $2 \leq$ `r13` < 3. Since the value of `r5` does not contain Xs (it is 16), the constraint of `r13` is updated by adding `r5`'s value (16) to the lower and upper bounds of `r13`'s constraints to produce the constraint: $18 \leq$ `r13` < 19. However, if the value of `r5` is 16'bXXXXX, the constraint on `r13` is updated to $3 \leq$ `r13` < 20, by adding the lower bounds and the upper bounds of the two constraints, respectively. This ensures that constraints are as tight as possible while encompassing all possible values.

3.3 Enforcing Constraints

Encoding and propagating constraints ensures that values of registers or memory locations that are constrained cannot go out of bounds of these constraints. To ensure this, we monitor all register and memory location values for changes during simulation. Whenever a register or a memory location is modified, we check its value against any constraint it has. If the value of the register or memory location could be out of bounds of the constraint, we enforce the constraint on the register or memory location by modifying its value appropriately. Our technique ensures that enforcing constraints does not eliminate exploration of any reachable states for a given application. A formal proof is presented in Section 3.4.

In addition to constraining memory and register values, it is important to ensure that memory *addresses* do not go out of bounds. In an indirect addressing mode, if the register holding the memory address contains Xs, there are several possible addresses that could be accessed. In such a case, the constraint on the register restricts the number of possible memory locations. While performing memory reads, all possible memory addresses (defined by the constrained conservative value) are read, and a conservative value is generated out of data read from memory. This value is sent to the data bus and used by the instruction. Similarly, while handling a memory write, both the address and the value could have Xs. In this case, we first resolve the constraint on the address by identifying the permissible locations for the element, based on the constraint and the value of the address. We then generate conservative values and update the constraints at all the resolved addresses. For instance, consider the instruction `mov r5, -5(r6)`. Assume that both `r5` and `r6` contain Xs. To handle proper execution of this instruction, we first obtain the constraint for `r6` and adjust the address constraint for `-5(r6)` according to the offset (i.e., `Lower_bound -5(r6)` ⟵ `Lower_bound (r6) - 5`) and `Upper_bound -5(r6)` ⟵ `Upper_bound (r6) - 5`). Then, for each address represented by `-5(r6)`'s value in the simulator (the value with the Xs), we check if the address is in the range of the constraint (i.e., `Lower_bound -5(r6)` < address < `Upper_bound -5(r6)`). For the addresses that are in the bound of the constraint, we write the conservative value of `r5` combined with the existing memory value to the locations pointed by the resolved addresses. This algorithm is presented in Algorithm 2.

3.4 Proof of CCS Correctness

Theorem 3.1 (Application Execution State Coverage). *Given a constraint $\boldsymbol{c}$ and an element (register/memory address) $\boldsymbol{e}$, enforcing $\boldsymbol{c}$ on $\boldsymbol{e}$ at a PC $\boldsymbol{p}$ does not eliminate exploration of any reachable states for application $\boldsymbol{A}$.*

Proof. Let $\boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_n}$ be consecutive conservative states generated at PC $\boldsymbol{p}$ by the Conservative State (**CS**) approach. By definition of conservative state, $\boldsymbol{S_1} \subset \boldsymbol{S_2} \subset \ldots \subset \boldsymbol{S_n}$. Let $\boldsymbol{S_i}$ be the first state where $\boldsymbol{e}$ violates $\boldsymbol{c}$. Thus, $\boldsymbol{S_i}$ covers all executions leading to $\boldsymbol{p}$ that have been explored until the i^{th} encounter of $\boldsymbol{p}$. I.e., for all states before $\boldsymbol{S_i}$ ($\boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_{i-1}}$), the Constrained Conservative State (**CCS**) approach and **CS** are identical. Since $\boldsymbol{S_i}$ violates $\boldsymbol{c}$, it necessarily covers some states that are not reachable by $\boldsymbol{A}$. Constraining $\boldsymbol{e}$ using $\boldsymbol{c}$ generates $\boldsymbol{S_i'}$ such that $\boldsymbol{S_i'}$ covers all possible values that $\boldsymbol{e}$ can assume in $\boldsymbol{A}$; only unreachable states are eliminated through the application of $\boldsymbol{c}$. Thus, continuing the simulation from $\boldsymbol{S_i'}$ will explore all valid states that are reachable by $\boldsymbol{A}$. □

Table 1: Benchmarks

Embedded Sensor Benchmarks [18]
mult, binSearch, div, inSort, tea8, rle, tHold, intAVG, intFilt
EEMBC Embedded Benchmarks [11]
AutoCorr, convEn, FFT, Viterbi
Complex Benchmarks
MergeSort , graph500 [1], highCC

4 EXPERIMENTAL SETUP

We perform evaluations on a silicon-proven openMSP430 [12] processor, synthesized, placed and routed in TSMC 65GP (65nm) technology using Synopsys Design Compiler and Cadence EDI System. The processor was implemented for an operating point of 1V and 100MHz. We implemented our constrained conservative state-based scalable symbolic co-analysis in a custom gate-level simulator that was built in-house in C++. We also developed a custom static binary analysis tool in Python for encoding constraints. The static constraints were stored in a JSON file and fed to the custom gate-level simulator, which the simulator uses for Propagation and Enforcement. We show results for all benchmarks from [18], all EEMBC benchmarks [11] that fit in the program memory of our processor, as well as complex and recursive benchmarks[1] designed to stress-test the scalability of our symbolic hardware-software co-analysis technique with complex control structures not found in the rest of our benchmarks (Table 1). Experiments are performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz operating frequency, 64GB RAM).

5 RESULTS

To illustrate the benefits of our proposed technique for symbolic co-analysis, we compare our *constrained conservative state* (CCS) symbolic co-analysis technique (Algorithm 1 **black**+**blue**+**red** text) against the *naive* symbolic co-analysis technique (Algorithm 1 **black** text only) and the state-of-the-art *conservative* symbolic co-analysis technique [8] (Algorithm 1 **black**+**blue** text). We compare analysis time and exercisable gate counts for the benchmarks described in Section 4. We show that the constrained conservative approach addresses the limitations of the naive and conservative approaches by yielding an exercisable gate count closer to the accurate naive approach, while also significantly reducing simulation time compared to the state-of-art with minimal overhead.

For benchmarks with simple control flow (i.e., no input-dependent branches), symbolic simulation only needs to consider a single execution path through the program; conservative states are never created, and the conservative and constrained conservative approaches will perform the same simulation as the naive approach. Since the results for these benchmarks (mult, intFilt, tea8, FFT, AutoCorr, convEn) do not show any variation between the simulation

[1] MergeSort is a recursive sorting algorithm. graph500 runs BFS on a graph. highCC (high Cyclomatic Complexity) is a synthetic benchmark that uses cyclic array accesses to alter the control flow of the application and has 16^{32} possible control flow paths.

Table 2: Constrained conservative state symbolic co-analysis reduces analysis time compared to naive and conservative state-based co-analysis and enables analysis of applications with complex control structures.

Benchmark	Analysis Time (Number of Simulation Cycles)				
	Naive	Consv.	CCS	%Reduction (w.r.t. Naive)	%Reduction (w.r.t. Consv.)
div	∞	186	178	-	4.30
intAVG	∞	337	329	-	2.37
rle	∞	7431	5951	-	19.92
rle_small	25496	6495	2153	91.56	66.85
binSearch	100468	9994	1551	98.46	84.48
tHold	20520	2615	1986	90.32	24.05
inSort	∞	22205	12120	-	45.42
inSort_small	24427	9106	5089	79.17	44.11
Viterbi	∞	69265	26389	-	61.90
MergeSort	∞	104574	16093	-	84.61
graph500	∞	185341	79663	-	57.02
highCC	∞	116290	80276	-	30.90

Table 3: Use of constraints reduces the number of explored symbolic execution paths.

Benchmark	Symbolic Execution Paths				
	Naive	Consv.	CCS	%Reduction (w.r.t. Naive)	%Reduction (w.r.t. Consv.
div	∞	9	7	-	22.22
intAVG	∞	15	13	-	13.33
rle	∞	129	101	-	21.71
rle_small	504	113	33	93.45	70.80
binSearch	2048	91	41	98.00	54.95
tHold	460	247	39	91.52	84.21
inSort	∞	121	67	-	44.63
inSort_small	476	115	65	86.34	43.48
Viterbi	∞	771	291	-	62.26
MergeSort	∞	1453	235	-	83.83
graph500	∞	1350	1124	-	16.74
highCC	∞	1604	756	-	52.80

Table 4: Use of constraints reduces the number of gates identified as exercisable.

Benchmark	Exercisable Gates Identified				
	Naive	Consv.	CCS	%Increase (w.r.t. Naive)	%Reduction (w.r.t. Consv.)
div	N/A †	3627	3566	-	1.68
intAVG	N/A †	3675	3648	-	0.73
rle	N/A †	4488	3759	-	16.24
rle_small	3185	4487	3740	17.43	16.65
binSearch	3065	3454	3424	11.71	0.87
tHold	2893	3530	3368	16.42	4.59
inSort	N/A †	5406	3518	-	34.92
inSort_small	3134	5418	3523	12.41	34.98
Viterbi	N/A †	5449	5449	-	0.00
MergeSort	N/A †	5134	4294	-	16.36
graph500	N/A †	5988	5987	-	0.02
highCC	N/A †	4007	3558	-	11.20

† Since these simulations did not finish, naive simulation would be forced to report that all 7218 gates of the design might be exercisable.

approaches and thus cannot be used to compare the techniques, we omit these benchmarks from our results tables due to space limitations. However, we did use these benchmarks to verify that the results for all three simulation approaches are consistent. Furthermore, our constrained conservative approach does not increase the execution time or number of execution paths considered.

Analysis Time. Table 2 compares analysis times for performing the symbolic simulation of each benchmark application. We use simulated clock cycles of the openMSP430 processor as a proxy of analysis time that is independent of the host computer's computational capability and load.[2] Constrained conservative analysis achieves the lowest analysis time for all benchmarks by effectively pruning the execution tree to eliminate consideration of already-visited states and states that are precluded by application constraints. For six of the benchmarks, naive symbolic simulation was not able to complete within 24 hours and was eventually killed after using all of our server's memory (64 GB RAM and 125 GB swap). These benchmarks are marked with ∞ in the naive column of Table 2. Meanwhile, the conservative state approach is able to analyze all of the benchmarks in under an hour. By applying application constraints on top of the conservative approach, CCS reduces analysis time for each benchmark, with a maximum reduction of 84.61% compared to the state-of-art conservative state approach. Applying software constraints to the symbolic simulation keeps conservative values within their legal ranges, significantly pruning the state space and resulting in a more efficient exploration of the application's possible states.

Table 3 shows the number of symbolic execution paths each symbolic simulation approach explores (as described in Section 2). In the conservative approach, new symbolic execution path subtrees are created at conditional branches and simulated if they have not been previously explored. By constraining the values of registers/memory elements in the processor, the constrained conservative approach reduces the number of symbolic execution paths that must be simulated to completely characterize all possible executions of an application. This significantly reduces analysis time for several applications. For MergeSort, an application with complex input-dependent control flow, the conservative state approach continues simulating symbolic execution paths until all bits of the loop iterator (for the loop that merges two sorted arrays) become Xs for a given recursive step. In the proposed constrained approach, simulation only proceeds until 6 Xs propagate into the loop iterator, since the maximum bound on the loop iterator is 34 (array size). The result is an 84% reduction of the number of symbolic execution paths that are explored and a corresponding 85% reduction in the number of analysis cycles. As processor complexity increases, the state space of the hardware-software symbolic co-analysis increases, and the potential benefits of constraining the symbolic simulation increase. E.g., a 64-bit processor has exponentially more possible states than a 16-bit processor, so the same loop bounds constraint applied to both would eliminate exponentially more states from consideration in a 64-bit processor vs. a 16-bit processor.

Exercisable Gates. Table 4 presents the count of exercisable gates reported by the three symbolic simulation approaches. All three approaches guarantee identification of all possible gates that can be exercised by any possible execution of an application; however, the approaches vary in their overestimation of the exercisable gates due to conservative state approximations. The naive approach does not use conservative states to cover multiple real states, and therefore, provides the most accurate report of the exercisable gate set. However, because naive simulation attempts to simulate all possible states of an application without approximation, naive simulation is not scalable and does not always complete. For some benchmark applications (e.g., inSort and rle), significantly reducing the input size (e.g., to 5 elements) reduces the size of the symbolic execution tree sufficiently to allow the naive approach to finish. We include

[2]The overhead introduced by the constrained conservative analysis indicated by red text in Algorithm 1 is between 1.1% and 1.9% per cycle.

small versions of those benchmarks in the results tables to enable further analysis and comparison of the simulation approaches.[3]

The conservative state approach identifies more exercisable gates than the naive approach. For applications with complex control flow, the overapproximation of the conservative state approach can be significant. The small versions of rle and inSort demonstrate that the conservative approach can significantly increase the number of gates marked as exercisable compared to naive symbolic simulation (e.g., 73% increase in exercisable gates reported for inSort_small). With the proposed constrained simulation, however, there is only a 12% increase in reported exercisable gates for the same application. Applying application constraints to the symbolic states avoids simulation of states that are not actually possible for the application and can significantly reduce the pessimism of applying conservative states to achieve a scalable symbolic simulation.

Compared to the conservative state approach, CCS reports fewer exercisable gates for all benchmarks, except Viterbi where the result is identical, with a maximum reduction of 35% (inSort). The static analysis used in this work generated a maximum of 7 constraints (for graph500) and a minimum of 1 constraint (for div). More sophisticated static analysis techniques may generate more constraints. Nevertheless, our work shows that even applying a small number of constraints can result in significant reduction of exercisable gates and analysis time compared to state-of-art conservative state symbolic co-analysis. The largest benefits come from benchmarks such as inSort, MergeSort, and rle, that access data using addresses containing Xs. This can potentially cause the address handler in openMSP430 to exercise all the peripherals, since they are in a unified address space. Constraining the addresses avoids this overapproximation of exercisable resources. Although binSearch also accesses data using addresses containing Xs, its structure already limits the number of Xs in addresses during conservative symbolic simulation, since the binSearch algorithm uses a right shift that guarantees that the upper 8 address bits are always zero. This reduces the exercisable gates reported by the conservative state approach for binSearch. Viterbi implements an iterative pointer chasing algorithm that involves many memory-accessing instructions. With the random memory access pattern of the application, the inputs of these instructions are all Xs, causing all the gates in the memory and peripheral path to be presumed exercisable. Constraints do help to restrict the *number* of memory accesses with unknown pointer values, since the accesses are made in a loop, and the loop bound can be determined by static analysis. This significantly reduces analysis time (by 62%) but does not help to reduce the exercisable gate count. Graph traversal in graph500 also involves a pointer chasing random memory access pattern. Similar to Viterbi, reduction in exercisable gates is negligible, but determining loop bounds via static analysis significantly reduces analysis time, by 57%.

6 CONCLUSION

Symbolic co-analysis of an application binary on the gate-level netlist of a processor can be used for application-specific power and energy optimizations. Although the state-of-the-art conservative symbolic co-analysis technique provides a scalable solution, we showed that this approach can be pessimistic both in terms of the exercisable gates and the number of symbolic execution paths explored. In this paper, we proposed a *constrained* conservative state symbolic hardware-software co-analysis technique that applies constraints to symbolic states to reduce the pessimism in marking gates as exercisable. In addition to guaranteeing identification of all possible exercisable gates for an application execution, the proposed technique significantly reduces simulation time and number of symbolic execution paths explored. We showed that our technique can reduce application analysis time by up to 84.61% while reducing the exercisable gates by up to 34.98% compared to the state-of-art conservative state symbolic co-analysis technique.

[3]Conservative symbolic simulations report slightly more exercisable gates for inSort_small than for inSort. At first, this seems counterintuitive; however, our analysis revealed that a few instructions were different between the two binaries. These instructions cause different gates to be exercised by each of the binaries. We confirmed that the additional exercisable gates in inSort_small trace back to instruction source/destination operand registers. These gates contribute to fewer than 0.2% of the total gates in the processor design and do not change the behavior of the core algorithm in the benchmarks.

REFERENCES

[1] [n.d.]. Graph 500. http://www.graph500.org.
[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.
[3] Valeria Bertacco, Maurizio Damiani, and Stefano Quer. 1999. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, 391–396.
[4] Randal E Bryant. 1991. Symbolic Simulation – Techniques and Applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. ACM, 517–521.
[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[6] Pankaj Chauhan, Edmund M Clarke, and Daniel Kroening. 2004. A SAT-based algorithm for reparameterization in symbolic simulation. In *Proceedings of the 41st annual Design Automation Conference*. ACM, 524–529.
[7] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori. 2017. Bespoke processors for applications with ultra-low area and power constraints. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 41–54.
[8] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Determining Application-Specific Peak Power and Energy Requirements for Ultra-Low-Power Processors. *ACM Trans. Comput. Syst.* 35, 3, Article 9 (Dec. 2017), 33 pages. https://doi.org/10.1145/3148052
[9] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Enabling Effective Module-oblivious Power Gating for Embedded Processors. In *High Performance Computer Architecture, 2017. HPCA 2017. IEEE 21st International Symposium on*. IEEE.
[10] Hari Cherupalli, Rakesh Kumar, and John Sartori. 2016. Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems. In *Computer Architecture (ISCA), 2016 43th Annual International Symposium on*. IEEE.
[11] EEMBC. 2020. EEMBC Benchmarks. http://www.eembc.org.
[12] O Girard. 2013. OpenMSP430 project. *available at opencores.org* (2013).
[13] A. Ibing and A. Mai. 2015. A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. 44–51.
[14] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. 2009. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In *2009 International Symposium on Code Generation and Optimization*. 136–146.
[15] Michele Magno, Luca Benini, Christian Spagnol, and E Popovici. 2013. Wearable low power dry surface wireless sensor node for healthcare monitoring application. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*. IEEE, 189–195.
[16] Chris Wilson, David L Dill, and Randal E Bryant. 2000. Symbolic simulation with approximate values. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 507–522.
[17] Ross Yu and Thomas Watteyne. 2013. Reliable, Low Power Wireless Sensor Networks for the Internet of Things: Making Wireless Sensors as Accessible as Web Servers. *Linear Technology* (2013). http://cds.linear.com/docs/en/white-paper/wp003.pdf
[18] Bo Zhai, Sanjay Pant, Leyla Nazhandali, Scott Hanson, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Todd Austin, Dennis Sylvester, et al. 2009. Energy-efficient subthreshold processor design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 17, 8 (2009), 1127–1137.
[19] W. Zuo, P. Li, D. Chen, L. Pouchet, Shunan Zhong, and J. Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10.