A scalable symbolic simulation tool for low power embedded systems

Subhash Sethumurugan†, Shashank Hegde†, Hari Cherupalli*, John Sartori†
† University of Minnesota and *Synopsys Inc., USA
{sethu018,hegde031,jsartori}@umn.edu,haricherupalli@gmail.com

Abstract

Recent work has demonstrated the effectiveness of using symbolic simulation to perform hardware software co-analysis on an application-processor pair and developed a variety of hardware and software design techniques and optimizations, ranging from providing system security guarantees to automated generation of application-specific bespoke processors. Despite their potential benefits, current state-of-the-art symbolic simulation tools for hardware-software co-analysis are restricted in their applicability, since prior work relies on a costly process of building a custom simulation tool for each processor design to be simulated. Furthermore, prior work does not describe how to extend the symbolic analysis technique to other processor designs.

In an effort to generalize the technique for any processor design, we propose a custom symbolic simulator that uses iverilog to perform symbolic behavioral simulation. With iverilog – an open source synthesis and simulation tool – we implement a design-agnostic symbolic simulation tool for hardware-software co-analysis. To demonstrate the generality of our tool, we apply symbolic analysis to three embedded processors with different ISAs: bm32 (a MIPS-based processor), darkRiscV (a RISC-V-based processor), and openMSP430 (based on MSP430). We use analysis results to generate bespoke processors for each design and observe gate count reductions of 27%, 16%, and 56% on these processors, respectively. Our results demonstrate the versatility of our simulation tool and the uniqueness of each design with respect to symbolic analysis and the bespoke methodology.

ACM Reference Format:
Subhash Sethumurugan†, Shashank Hegde†, Hari Cherupalli*, John Sartori† . 2022. A scalable symbolic simulation tool for low power embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530433

1 Introduction

Emerging applications like wearables [17, 22], implantables [12, 21], and IoT applications [14, 16, 19, 20, 23, 30] are characterised by ultra-low area and power requirements. This is because many of these systems are powered by battery or energy harvesting [14, 17] and have form factor restrictions [19] due to the nature of the applications. Another defining characteristic of these ultra-low-power (ULP) applications is that they tend to run the same software over and over, as defined by their application. A body of recent work has shown that using a novel hardware-software co-analysis technique based on symbolic simulation can learn application-specific hardware behaviors and enable significant power, energy, and area savings for ultra-low-power systems [4–6, 8]. Further, the co-analysis technique has also been shown to enable security guarantees for ultra-low-power systems [7], demonstrating its applicability beyond energy and area optimizations.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530433

Despite the significant potential of application-specific design and optimization techniques, applicability has been limited, since the symbolic co-analysis tools developed in previous works were developed for a single processor (openMSP430), and extending them to analyze and optimize other processor designs or architectures requires the challenging and time-consuming task of developing a new custom simulation tool for each new design. This simulation approach is not scalable, especially for industry, as each application may use a different design, and it is infeasible to write a custom simulation tool for each design. In this work, we overcome the limitations of prior work and demonstrate a general, automated tool for hardware-software co-analysis that can analyze any processor design and enable the benefits of application-specific design and optimization.

A general purpose microprocessor is designed to run any application, and thus, contains more gates than any specific application may exercise during execution. The symbolic hardware-software co-analysis methodology proposed in prior works performs a symbolic simulation of a system's software application on the system's hardware, in which inputs to the application are represented as symbols (Xs) that represent an unknown logic value. The resulting simulation characterizes the behavior of the system hardware for all possible executions of the system software, for all possible inputs to the software. One of the primary outputs generated by the symbolic analysis is a dichotomy of the processor's logic into two sets of gates – one set of gates that could be exercised by the application in some execution and another set of gates that are guaranteed to never be exercised for any possible execution of the application. Both of these sets are used in prior work for various application-specific design and optimization techniques. For example, the set of exercisable gates, along with their activity profiles, can be used to perform application-specific power gating [6], determine application-specific peak power and energy requirements [5], and provide security guarantees for an embedded system [7]. The set of gates that can never be exercised can enable application-specific voltage overscaling [8, 18], power gating [6], and automatic generation of application-specific bespoke processors [4].

Prior works perform symbolic simulation on the gate-level netlist of the processor to perform hardware-software co-analysis and identify the sets of exercisable and unexercisable gates. In the simulation, all inputs are set to Xs and propagated through the gate-level netlist. If an X propagates to a gate, it is considered exercisable, since for some input the gate could toggle. When X propagates to the condition of a conditional branch instruction, the simulation must simulate all possible execution paths from the branch. In several applications, especially those with complex control flow structures, enumerating and simulating all possible execution paths is intractable. Thus, to enable a scalable simulation approach for complex applications, prior works propose generating and simulating *conservative states* that represent all the states observed at each PC-changing instruction [4, 15]. One conservative state can represent multiple observed states (e.g., XX represents 00, 01, 10, and 11), and thus, simulating from a conservative state can enable the simulation to converge faster while guaranteeing coverage of all possible application states, at the possible cost of some over-approximation of the observed states.

Leveraging the application-specific design information generated by symbolic hardware-software co-analysis, prior works have demonstrated several analysis, design, and optimization techniques that show significant area, energy, and security benefits [4–8]. However, demonstrated benefits have been restricted to a single processor, since

the co-analysis tool used in prior works was custom designed for the openMSP430 processor, and performing co-analysis on a different processor would require development of a custom simulation tool for each new design. Modern gate-level simulators such as VCS [26] can perform cycle accurate simulations; however, they do not support all features necessary to run hardware-software co-analysis. For example, modern simulators do not support custom propagation of symbols, management of conservative states, and splitting the simulation on observing a particular symbolic signal. In our work, we develop a design-agnostic simulation tool that performs symbolic hardware-software co-analysis with cycle-accurate precision at the gate level. We extend an open-source design synthesis and simulation tool – iverilog – to support symbolic simulations and enable the use of conservative gate-level execution states. In this paper, we describe how we extended iverilog to support symbolic hardware-software co-analysis for an arbitrary digital design. We also demonstrate the generality of our tool by performing symbolic co-analysis for three microprocessors with different ISAs – MSP430 [11], MIPS [24], and RISC-V [9]. We demonstrate that the results of symbolic co-analysis can be used to perform application-specific design and optimization for these processors by generating bespoke processors [4] for various embedded applications, and we report the exercisable gate count, number of execution paths evaluated, and simulation time for analyzing the applications on the microprocessor designs.

2 Related work

Prior works on application-specific system design and optimization propose symbolic hardware-software co-analysis and demonstrate its use in a number of applications, from providing security guarantees in embedded systems [7], to performing application-specific optimizations that reduce power and energy without sacrificing performance or functionality [5, 6, 8, 18], to automatically generating application-specific bespoke processors for ultra-low-power embedded systems [4]. However, prior works rely on developing a custom simulator for each processor to be analyzed and optimized. Since this is a challenging and time-consuming endeavor that is not scalable, prior works only demonstrated results for a single processor (openMSP430). In our work, we develop a design-agnostic symbolic simulation tool that can apply symbolic hardware-software co-analysis techniques to any digital design and application. Our tool offers a scalable approach to easily extend symbolic analysis and subsequently enable application-specific optimization for new designs.

Prior work on property-driven automatic hardware transformation [1] developed a property-driven framework for automatically generating hardware for a reduced ISA, where a specified list of instructions or ISA features are not supported. The work uses a property library to annotate all gates in the design and performs property checking to identify gates for which the properties are verified. Developing a property library that encodes ISA restrictions for each application is a manual process that can be both challenging and time-consuming. Our symbolic simulation tool, on the other hand, can easily analyze a new design with minimal user effort or expertise. Further, our tool is able to handle designs in any format – RTL or gate-level netlist – described in any hardware description language, e.g., verilog, VHDL, or system verilog.

In our work, we discuss saving and restoring simulation state in iverilog. Restoring simulation state involves assigning values to design elements, such as nets and registers. Prior works have used verilog constructs such as `force` and `release` for fault injection in design elements [10]. However, at any simulation point, `force` and `release` allow us to assign only one value to a design element. To assign a different value, the testbench must be modified and recompiled. Also, the simulation must be restarted from the beginning. By saving and restoring simulation states, we avoid this overhead. Using `force` and `release`, we cannot split the simulation and launch multiple instances. Our approach allows us to parallelize simulations for different execution paths.

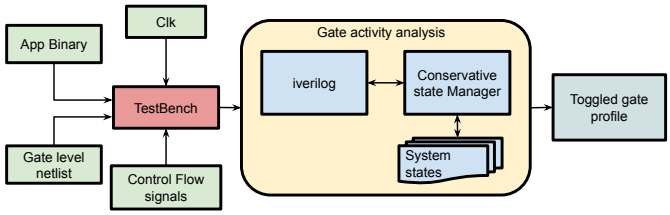


Figure 1. Our processor-agnostic symbolic co-analysis tool is built on top of iverilog to allow hardware-software co-analysis of any digital design.

3 A generic tool for hardware-software co-analysis

Prior works have demonstrated the effectiveness of symbolic hardware-software co-analysis for a variety of analysis, design, and optimization techniques. While the methodology itself is generic, the tool developed in prior works was custom built for a single processor and hence was not generic. In this section, we discuss the implementation of a symbolic hardware-software co-analysis tool in iverilog [29], an open-source verilog synthesis and simulation tool. By implementing the technique within the framework of a generic verilog simulator, we enable the tool to easily analyze different designs with minimal user effort or expertise required.

Performing the symbolic hardware-software co-analysis of an application on a microprocessor design involves performing a gate-level simulation in which all application inputs are replaced by symbols (X) indicating unknown logic, thus simulating the behavior of the microprocessor for all possible application inputs. When an X is propagated to an instruction that affects control flow (e.g., branch, jump), multiple simulations are spawned to cover all possible execution paths of the application from the instruction. To handle execution path explosion in complex applications, we follow the approach of using a conservative state to represent all execution states observed at the same program counter (PC) [10]. This guarantees coverage of all possible execution states while allowing the simulations to converge. To accommodate symbolic co-analysis, we make the following modifications to iverilog source code.

1) *Monitor critical microprocessor signals*: To identify when X propagates to an instruction that affects control flow, we implement a system task function called `monitor_x()` in iverilog that monitors a list of signals. For example, the signals could be a combination of the ALU flags, like N, Z, C, and V (negative, zero, carry, and overflow) that determine the result of a conditional branch instruction that indicates if a branch is taken.

2) *Save the simulation state*: To cover all possible executions from a branch with unknown outcome, we first dump the simulation state before the execution of the instruction that affects control flow. The simulation state indicates the state of the microprocessor along with the state of the simulator (e.g., the event queue).

3) *Continue simulation from a saved state*: To simulate all possible executions from the instruction affecting control flow, we make multiple copies of the saved simulation state and modify each copy with the status that allows the microprocessor to take one of the possible executions. We enhance the simulator to read the modified simulation state and continue the simulation from the halted state. For this, we implement another system task called `initialize_state()`.

Figure 1 illustrates the entire simulation flow. To perform symbolic co-analysis, the user provides the application binary and the gate-level netlist to a testbench harness, along with a list of control flow signals to monitor. The testbench instantiates the design, loads the application binary, and provides inputs (Xs) to the application. The testbench also calls the `monitor_x()` system task, providing the user-specified control flow signals as argument. iverilog assimilates all the information into an iverilog-specific intermediate representation (vvp assembly) [29] and starts the simulation. Once the

simulation reaches a PC-changing instruction where any of the signals that determine control flow are X, the execution path becomes non-deterministic, and we must explore all possible execution paths. At this point, the simulation is halted, the simulation state is saved, and the Conservative State Manager (CSM) is alerted. The CSM is a program that maintains a repository of previously-simulated states. A simulation state is indexed by the PC of the PC-changing instruction at which it was observed. When the simulator halts the simulation and provides the simulation state to the CSM, the CSM compares the state with the most conservative state that has been simulated thus far for the same PC. If the current state is a strict subset of the previously-simulated state, this state has already been evaluated, and hence, further simulation is not required. If the current state is not a strict subset, the CSM generates a more conservative state that covers both states by merging the current state and existing conservative state. Once the new conservative state is formed, appropriate control flow signals are set to continue down the possible execution paths from the PC-changing instruction. Algorithm 1 describes the simulation procedure.

Algorithm 1 Symbolic Hardware-Software Co-analysis using iverilog

```
1. Procedure symbolic_simulation(app_binary, design_netlist, control_signals)
2. Load the design_netlist and initialize the Memory.
3. Load app_binary into program memory
4. Propagate reset signal
5. s ← State at start of app_binary
6. cs ← control_signals
7. Table of previously observed symbolic states, T.insert(s)
8. Stack of un-processed execution paths, U.push(s)
9. T_p ← ϕ // Initialize empty toggle profile
10. T_n ← ϕ // Initialize empty toggle nets
11. while U != ∅ do
12.     e ← U.pop()
13.     e.set_control_signals() // set control signals for a execution path
14.     $initialize_state(e)
15.     // halt if any of the control signal becomes X
16.     while $monitor_x(cs) == 0 do
17.         e' ← propagate_gate_values(e) // simulate this cycle
18.         e ← e' // advance cycle state
19.     end while
20.     c ← T.get_conservative_state(e)
21.     if e' ⊄ c then
22.         e'' ← T.make_conservative_superstate(c,e')
23.         U.push(e'')
24.         T_p.save_toggle_profiles(e'')
25.     else
26.         break
27.     end if
28. end while
29. // Merge toggled nets of all the toggled paths.
30. foreach p ∈ T_p do
31.     T_n.append(p)
32. end for
33. // Mark driver gates of the corresponding nets as toggled.
34. foreach n ∈ T_n do
35.     if n.toggled() then
36.         g ← n.getDriverGate()
37.         g.setToggled()
38.     end if
39. end for
40. foreach g ∈ design_netlist do
41.     if g.untoggled then
42.         annotate_constant_value(g,s) // record the gate's initial (and final) value
43.     end if
44. end for
```

The simulation is complete when there are no new states to simulate. We then obtain gate activity information for all explored paths. We combine the activity information to generate the gate activity information for the entire application. The gate activity information indicates all the gates that are exercisable by the application. This information can be used for subsequent application-specific design optimizations. For example, to generate a bespoke processor, unexercisable gates are pruned away and the microprocessor design is re-synthesized to generate a new gate-level netlist with lower area and power consumption. During re-synthesis, fanout values of pruned gates are set to the constant value seen during the symbolic simulation of the target application. In Section 5 we demonstrate the generality of the novel analysis tool by evaluating our methodology on three different processor implementations, each based on a different ISA.

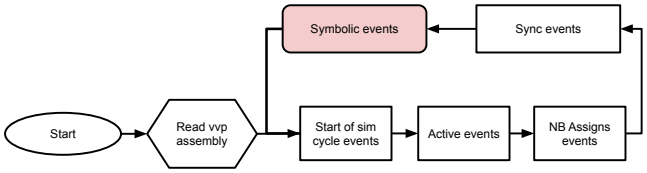


Figure 2. We add a new type of event to capture 'symbolic events' in iverilog's event queue. This enables us to monitor control signals for X and halt the simulation when necessary. The VVP engine is a part of iverilog source that executes an iverilog compiled assembly code that is generated from the verilog testbench.

3.1 iverilog software flow enhancement

iverilog is an event-driven simulator, where a set of events represents a time step. Upon the execution of these events, the simulation time progresses. Events are categorized into five event regions, and each region represents a similar set of events. The event regions are executed in the order shown in Figure 2. Since we implement symbolic simulation as a plug-in feature to iverilog, we ensure that our modifications do not affect the existing flow. Therefore, we create a new event region called *Symbolic events* and execute them after the other event regions. Symbolic events includes monitoring control flow signals, halting the simulation when X is detected, serializing and saving the processor and simulator state, and restarting the simulation from a saved state. By executing symbolic events last, we ensure that all events for the time step have already executed. When the simulation restarts, there may be a few events not belonging to the symbolic events region that are executed before initialization. However, the state initialization in the symbolic events region overrides the entire simulator and processor state. This nullifies the effects of any event executed before initialization. As this override occurs only in the first time step, the overhead of this process is minimal.

3.2 Designing a testbench for symbolic hardware-software co-analysis for iverilog

Listing 1 describes a simple testbench that uses the symbolic simulation feature of the iverilog tool. The user must follow the steps described below to perform symbolic hardware-software co-analysis.
1) The testbench calls two system tasks: `monitor_x()` and `initialization_state()` in an initial block. `monitor_x()` accepts a list of signals that affect control flow as argument, allowing iverilog to halt simulation when the execution path is non-deterministic. `initialization_state()` accepts simulation state as argument to allow iverilog to initialize the processor and simulator states, and begin simulation from a previously halted state.
2) The testbench must instantiate and reset the processor.
3) The testbench must initialize the processor inputs – registers and memory – to Xs to allow iverilog to simulate all possible execution paths of the application.

3.3 Conservative State Management

Simulation halts if one or more Xs is encountered in a *monitored* state variable or if the simulation terminating condition is met, indicating that all possible application states have been simulated. In case of an X in a monitored signal, we launch multiple instances of iverilog that execute the branches of the simulation where the Xs in the monitored state are re-interpreted as ones or zeros to cover all legal scenarios. Alternatively, we can apply the conservative state optimization proposed in prior works [4]. Using this optimization, a more conservative state of the saved state is generated by merging all the previously-observed states that match the PC of the current saved state. Applying the conservative state optimization significantly accelerates simulation by allowing many simulation paths that are covered by the conservative state to be discarded.

How conservative states are formed can be configured in the simulator. A designer can choose any approach to form conservative

Listing 1. Simple verilog test bench harness for starting symbolic simulation

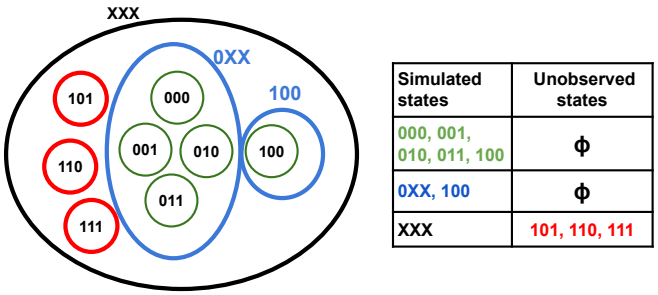
```
initial
begin
    $monitor_x("control_signals.ini");
    $initialize_state("sim_state.log");

    RST_n = 1'b0;
    #100 RST_n = 1'b1;
end

reg [7:0]data_memory[7999:0]; // 8kB data memory

// Instantiate Design.
GateLevelNetList dut(input reg1, reg2,..., data_memory);

initial
begin
    reg1 = 16{1'bx};
    reg2 = 16{1'bx};
    // set input dependent memory locations as X
    for (i = start_loc; i < end_loc; i = i + 1)
    begin
        data_memory = 8'bxxxxxxxx;
    end
end
... // other necessary initializations
```



Simulated states	Unobserved states
000, 001, 010, 011, 100	ϕ
0XX, 100	ϕ
XXX	101, 110, 111

Figure 3. Various approaches for conservative state generation exhibit trade-offs between simulation effort and conservative over-approximation. To capture all states in the first row (green) we could either create two conservative states as shown in the second row (blue) or one uber-conservative state as shown in the third row (red).

states, depending on convergence and accuracy requirements, as long as the approach ensures that the formed conservative state covers all observed states. For example, the approach used in prior work is to generate a single conservative state by merging simulation states and replacing all differing bits with Xs. Generating a single state to cover all observed states allows the simulation to converge the quickest and is most scalable, but it is also the most conservative, and represents some gates as exercisable that may not actually be exercisable. Consider the in Figure 3, where the observed states for a given PC are represented by the green circles. A conservative state of **XXX** encompasses all the observed states, and in addition, covers a few unobserved states. Though this approach reduces simulation time significantly, it can lead to over-approximation of exercisable gates. As another example, consider using a conservative state of **0XX** along with the state **100**, represented by blue circles. This conservative state formulation requires simulation of two execution paths rather than the original five and avoids representing unobserved states. In our tool, the CSM supports the ability to specify a custom conservative state generation approach by providing the rules of conservative state generation. Another example of a custom approach could be using application constraints to constrain conservative states [15]. The CSM accepts constraints in the form of a text file and uses them to reduce over-approximation of conservative states.

The CSM keeps track of all the saved states along with their PC values and generates conservative states to be fed into the next branch in the simulation. CSM is also responsible for triggering the launch of the iverilog instance that simulates the next branch. Since each branch of the simulation can be run by a separate process, launching these processes in parallel can drastically improve simulation time.

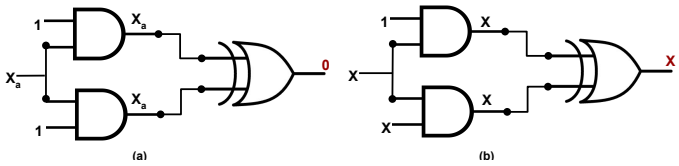


Figure 4. Our symbolic tool allows rules for symbol propagation to be customized. The left sub-figure shows a case where circuit inputs are propagated as separate symbolic values, while the right sub-figure shows a case where the symbolic values carry no identifying information and thus cannot be distinguished.

3.4 Propagation of symbols

The simulation tool also allows customization of symbol propagation. Different approaches for propagating Xs are used for different application-specific optimizations. For example, optimizations that require the identification of unexercisable gates must track the propagation of Xs, as this indicates the possibility of a gate being exercised for some application input, while to provide security guarantees, symbols must also propagate taint information [7]. For a less conservative simulation, we may want to track the propagation of each unknown value individually. This can allow simplification when the same symbol recombines at a gate.

For example, the left sub-figure of Figure 4 shows a case where inputs to the circuit are propagated as separate symbolic values. In this case, it can be determined that the inputs to the XOR gate have the same unknown value, and the output of the XOR gate is logic 0. In the right sub-figure, no identifying information is propagated with the symbols, so it cannot be determined that the inputs to the XOR gate have the same value, and the output must be assumed to be unknown (X). The latter approach is easier and more scalable to simulate, while the former is less conservative.

4 Methodology

We verify our technique on a silicon-proven processor – openMSP430 [13] – an open-source version of one of the most popular ULP processors [2, 28], a custom implementation of an open-source 32-bit MIPS processor – bm32 [24] – and DarkRISCV SoC [9], a RISCV implementation that implements the RV32e ISA [27] with integer registers reduced to 16 bits. Our implementation of DarkRISCV only modeled the processor core and memory. The processor designs are synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [25] and Cadence EDI System [3].

Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using our symbolic simulation tool. Table 1 lists our benchmark applications. We show results for the benchmarks that fit in the program memory of the processors. Table 2 lists the selected processors and their features.

The gate-level simulations were performed using an enhanced version of iverilog [29] written in C++ and a Conservative State Manager written in Perl. The CSM uses the conservative state approach used in prior work [4].

Benchmarks are chosen to be representative of emerging ULP application domains such as wearables, internet of things, and sensor networks [31]. Also, benchmarks were selected to represent a range of complexity in terms of control flow and execution length. Experiments were performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz frequency, 64GB RAM).

5 Results

In this section, we present and discuss the results of evaluating three microprocessor designs – openMSP430 (MSP430), bm32 (MIPS32), and dr5 (RV32e), using our tool. We run conservative-state based symbolic simulation for all the applications in Table 1 and generate

Table 1. Benchmark Applications

Benchmark	Description
Div	Unsigned integer division
inSort	in-place insertion sort
binSearch	Binary search
tHold	Digital threshold detector
mult	unsigned multiplication
tea8	TEA encryption algorithm

Table 2. Target Platform Characterization

Design	ISA	Features
bm32	MIPS32	32-bit MIPS implementation, with hardware multiplier.
openMSP430	MSP430	16bit microcontroller with 16x16 Hardware Multiplier, Watchdog, GPIO, TimerA
dr5	RV32e	32-bit RISCV embedded ISA with 16 integer register, 3 stage pipeline.

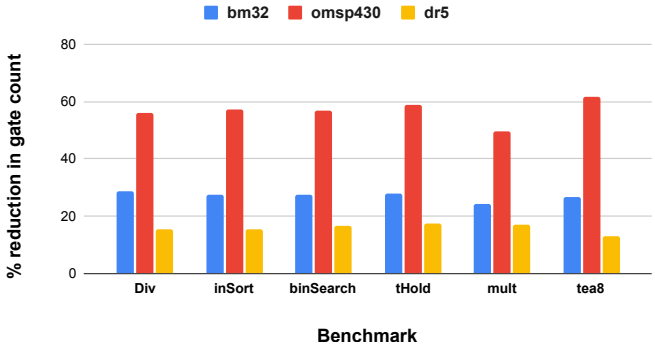


Figure 5. Benchmarks run on MSP430 processor have a higher reduction in exercisable gate count compared to MIPS and RISCV processors because of the presence of unused peripherals in MSP430.

the input-independent gate activity profile. We then prune away the unused gates and re-synthesize the design to generate an area and energy efficient *bespoke* processor, as in [4]. We present the results of our analysis using two metrics – exercisable gate count and the number of simulation paths.

5.0.1 Validation. To verify that the bespoke netlist generated with our generalized simulation tool works correctly, we simulate the application behavior using fixed known inputs on both the original and the bespoke gate-level netlist. We verified that the outputs from both the designs are the same. We also verified that the set of exercised gates for the fixed input run is a *subset* of the set of exercisable gates reported by our tool. Also, to ensure that the bespoke optimization enhancements made to iverilog do not affect the existing simulation capabilities, we verified that the event list from the baseline iverilog version matches the iverilog version after our enhancements at simulation points for applications that are picked at random.

5.0.2 Exercisable Gates. Table 3 shows the number of gates marked as exercisable by an application for the three designs. The total number of gates in the three microprocessor designs – bm32, openMSP30, dr5 – are 16795, 7218, and 7578, respectively. Using our tool to perform symbolic hardware-software co-analysis, we achieve a gate count reduction of 27%, 56% and 16% for these processors, respectively. Figure 5 shows the percentage reduction of the toggled gates for all benchmarks in Table 1. We observe that designs with external peripherals tend to have a higher gate count reduction. This is because, for applications that do not use peripherals, the set of gates representing the peripheral logic will not be exercised and can be safely removed. Since dr5 does not contain any peripheral logic such as a multiplier, it exhibits a relatively smaller reduction in the toggled gate count.

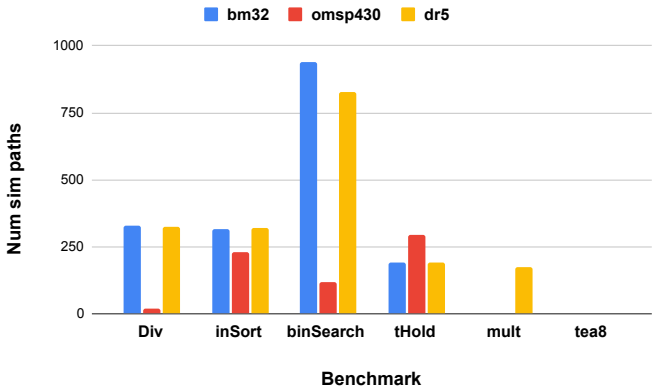


Figure 6. Benchmarks run on MIPS and RISCV processors have a higher number of simulated paths because a 16-bit register is used to indicate branch conditions, whereas in MSP430, a 1-bit register is used, resulting in fewer conservative states.

5.0.3 Simulation paths. From the simulation paths reported in Figure 6, we observe that bm32 and dr5 require significantly more simulation paths than openMSP430 to complete symbolic simulation. This is because of a fundamental difference in how `compare` instructions are implemented in the designs and how that affects conditional jumps in an application. In openMSP430, the result of the `compare` instruction is stored in program status word in the form of N, Z, C, and V flags. Based on the value of these flags (1 or 0), conditional jumps are resolved. In bm32 and dr5, on the other hand, the `compare` instruction is implemented as a subtraction operation, and the resulting value is stored in a 16-bit register, which is used to resolve conditional jumps. As discussed in Section 3 we halt the simulation when the output of a `compare` instruction preceding a conditional jump resolves to one or more Xs. In the case of openMSP430, this means when any of the NZCV flags of the status register is an X. In the case of bm32 and dr5, this means that the 16-bit register that holds the result of subtraction contains one or more Xs. If the 16-bit result register already contains an X, subsequent subtractions (such as `compare` used to evaluate loop termination conditions) would increase the number of Xs in the register. In most applications, all possible execution paths are only evaluated when the entire register fills with Xs. This significantly increases the number of paths that need to be evaluated for bm32 and dr5 processors. Since the NZCV flags in openMSP430 are 1-bit each, there are no additional Xs incurred at every `compare` instruction. This means that openMSP430 is able to converge faster, while for bm32 and dr5, several simulation instances are necessary to reach a simulation state that represents all possible subtraction operations. Due to the use of status bits (NZCV flags), benchmarks compiled for openMSP430 also have fewer conditional branch instructions compared to benchmarks in other processors, leading to fewer explored paths.

Another factor that significantly affected the simulation time for dr5 is the lack of a hardware multiplier module. As such, the compiler for dr5 performs multiplication in software using a library implementation of multiplication in the form of repeated additions in a loop. This leads to the use of input-dependent conditional branches to perform multiplication in dr5. Since input-dependent conditional branches lead to the generation of multiple simulation paths, we see that for the benchmark mult, dr5 has more than one simulation path in Figure 6, while the number of simulation paths for the other two processors that use a hardware multiplier is one.

Finally, Figure 6 shows that for the benchmark tHold, the number of simulated paths is higher for openMSP430 compared to bm32 and dr5, contradicting the trend seen in the other benchmarks. This is because the compiled binary for openMSP430 had three conditional branch instructions vs two in dr5 and bm32. Hence, in openMSP430, the number of execution split points in each loop iteration of tHold is three, compared to only two for dr5 and bm32. This difference

Table 3. Gate count analysis

Benchmark	BM32 tgc: 16795		omsp430 tgc: 7218		darkriscv tgc: 7578	
	GateCount	% reduction	GateCount	% reduction	GateCount	% reduction
Div	12008	28.5	3175	56.01	6399	15.56
inSort	12210	27.3	3098	57.08	6402	15.52
binSearch	12200	27.36	3115	56.84	6324	16.55
tHold	12139	27.72	2970	58.85	6259	17.41
mult	12707	24.34	3651	49.42	6299	16.88
tea8	12340	26.53	2755	61.83	6577	13.21

Table 4. Simulation path and runtime analysis

Benchmark	BM32 tgc: 16795			omsp430 tgc: 7218			darkriscv tgc: 7578		
	paths created	skipped	simulated cycles	paths created	skipped	simulated cycles	paths created	skipped	simulated cycles
Div	327	112	53202	17	8	776	325	112	13149
inSort	315	130	35044	230	118	18086	319	132	9382
binSearch	941	190	154198	119	62	9715	829	190	2374
tHold	191	68	17168	293	184	13030	191	68	4690
mult	1	0	528	1	0	258	175	60	5790
tea8	1	0	10018	1	0	3852	1	0	4534

quickly adds up as the symbolic execution tree is built, leading to a higher number of simulation paths for openMSP430. Table 4 provides the number of simulation paths created and skipped, along with the simulated cycles for each application in all the designs.

6 Conclusion

Current state-of-the-art symbolic simulation tools for hardware-software co-analysis are restricted in their applicability, since prior work relies on a costly process of building a custom simulation tool for each processor design. Furthermore, prior work does not describe how to extend the symbolic analysis technique to other processor designs. In this paper, we described how we modified iverilog to support propagation of symbolic values, conservative state generation and simulation, monitoring of critical control signals, and saving and restoration of simulation states, thus creating a design-agnostic symbolic simulation tool for hardware-software co-analysis. We demonstrated the generality of our tool by performing symbolic analysis on three embedded processors with different ISAs, and we also used analysis results from our tool to generate bespoke processors for each processor design and discussed the impact of architectures on the results and simulation times. Our results demonstrate the versatility of our simulation tool and the uniqueness of each design with respect to symbolic analysis and the bespoke methodology.

References

[1] Nathan Bleier, John Sartori, and Rakesh Kumar. 2021. Property-driven Automatic Generation of Reduced-ISA Hardware. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 349–354.

[2] Jacob Borgeson. 2012. Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform. *Texas Instruments White Paper* (2012). http://www.ti.com/lit/wp/slay019a/slay019a.pdf

[3] Cadence. [n.d.]. *Encounter User Guide.* http://www.cadence.com/

[4] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John 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. https://doi.org/10.1145/3079856.3080247

[5] 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. on Computer Systems (TOCS)* 35, 3 (2017), 1–33.

[6] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Enabling Effective Module-oblivious Power Gating for Embedded Processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 157–168.

[7] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based gate-level information flow security for IoT systems. In *Procs. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 328–340.

[8] 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.

[9] darklife. 2021. DarkRISCV open source riscv implementation. (2021). https://github.com/darklife/darkriscv

[10] Sunil R Das, Sujoy Mukherjee, Emil M Petriu, Mansour H Assaf, Mehmet Sahinoglu, and Wen-Ben Jone. 2006. An improved fault simulation approach based on verilog with application to ISCAS benchmark circuits. In *2006 IEEE Instrumentation and Measurement Technology Conference Proceedings*. IEEE, 1902–1907.

[11] Olivier Gerard. 2018. openMSP430, a synthesizable 16bit microcontroller core written in Verilog. (2018). https://opencores.org/projects/openmsp430

[12] Paul Gerrish, Erik Herrmann, Larry Tyler, and Kevin Walsh. 2005. Challenges and constraints in designing implantable medical ICs. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 435–444.

[13] O Girard. 2013. OpenMSP430 project. *available at opencores.org* (2013).

[14] G. Hackmann, Weijun Guo, Guirong Yan, Zhuoxiong Sun, Chenyang Lu, and S. Dyke. 2014. Cyber-Physical Codesign of Distributed Structural Health Monitoring with Wireless Sensor Networks. *Parallel and Distributed Systems, IEEE Transactions on* 25, 1 (Jan 2014), 63–72. https://doi.org/10.1109/TPDS.2013.30

[15] 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 *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 318–324.

[16] BK Charlotte Kjellander, Wiljan TT Smaal, Kris Myny, Jan Genoe, Wim Dehaene, Paul Heremans, and Gerwin H Gelinck. 2013. Optimized circuit design for flexible 8-bit RFID transponders with active layer of ink-jet printed small molecule semiconductors. *Organic Electronics* 14, 3 (2013), 768–774.

[17] 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.

[18] Veni Mohan, Akhilesh Iyer, and John Sartori. 2018. Enhancing Workload-dependent Voltage Scaling for Energy-efficient Ultra-low-power Embedded Systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6.

[19] Kris Myny, Steve Smout, Maarten Rockelé, Ajay Bhoolokam, Tung Huei Ke, Soeren Steudel, Brian Cobb, Aashini Gulati, Francisco Gonzalez Rodriguez, Koji Obata, et al. 2014. A thin-film microprocessor with inkjet print-programmable memory. *Scientific reports* 4 (2014), 7398.

[20] K. Myny, E. van Veenendaal, G. H. Gelinck, J. Genoe, W. Dehaene, and P. Heremans. 2011. An 8b organic microprocessor on plastic foil. In *2011 IEEE International Solid-State Circuits Conference*. 322–324. https://doi.org/10.1109/ISSCC.2011.5746337

[21] Seetharam Narasimhan, Hillel J Chiel, and Swarup Bhunia. 2011. Ultra-low-power and robust digital-signal-processing hardware for implantable neural interface microsystems. *IEEE trans. on biomedical circuits and systems* 5, 2 (2011), 169–178.

[22] Chulsung Park, Pai H Chou, Ying Bai, Robert Matthews, and Andrew Hibbs. 2006. An ultra-wearable, wireless, low power ECG monitoring system. In *Biomedical Circuits and Systems Conference, 2006. BioCAS 2006. IEEE*. IEEE, 241–244.

[23] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2012. Mementos: system support for long-running computation on RFID-scale devices. *Acm Sigplan Notices* 47, 4 (2012), 159–170.

[24] C. Roth, L.K. John, and B.K. Lee. 2015. *Digital Systems Design Using Verilog*. Cengage Learning. https://books.google.com/books?id=Qi1BBAAAQBAJ

[25] Synopsys. [n.d.]. *Design Compiler User Guide.* http://www.synopsys.com/

[26] Synopsys. [n.d.]. *VCS/VCSi User Guide.* http://www.synopsys.com/

[27] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[28] Wikipedia. 2016. List of wireless sensor nodes. https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes [Online; accessed 7-April-2016].

[29] Stephen Williams and Michael Baxter. 2002. Icarus Verilog: Open-Source Verilog More than a Year Later. *Linux J.* 2002, 99 (jul 2002), 3.

[30] 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

[31] 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.