Optimizing Blocks in an SoC Using Symbolic Code-Statement Reachability Analysis

Hong-Zu Chou†, Kai-Hui Chang‡, and Sy-Yen Kuo†

†Electrical Engineering Department, National Taiwan University, Taipei, Taiwan

‡Avery Design Systems, Inc., Andover, MA, USA

sykuo@cc.ee.ntu.edu.tw

Abstract— Optimizing blocks in a System-on-Chip (SoC) circuit is becoming more and more important nowadays due to the use of third-party Intellectual Properties (IPs) and reused design blocks. In this paper, we propose techniques and methodologies that utilize abundant external don't-cares that exist in an SoC environment for block optimization. Our symbolic code-statement reachability analysis can extract don't-care conditions from constrained-random testbenches or other design blocks to identify unreachable conditional blocks in the design code. Those blocks can then be removed before logic synthesis is performed to produce smaller and more power-efficient final circuits. Our results show that we can optimize designs under different constraints and provide additional flexibility for SoC design flows.

I. Introduction

The use of Intellectual Properties (IPs) in System-on-Chip (SoC) circuits has become a common design practice recently to accelerate the circuit design process. Design blocks are also often reused in new chips to reduce design effort. However, these approaches may leave unnecessarily large design blocks in the final chip because the blocks may be over-provisioned with respect to the target functionality. As a result, power efficiency and performance are degraded. To solve such problems, optimizations should be applied to remove the unused logic from the reused design blocks. Such optimizations can be achieved by utilizing don't-care conditions that exist due to the surrounding environment. The use of external instead of internal don't-cares makes the optimization of IPs quite different from traditional synthesis optimizations. Although techniques exist to utilize such optimization opportunities [3], these techniques work on the gate level only. Not being able to work on the higher level code greatly limits the optimization power of such techniques because removing an unused high-level code block may eliminate thousands of gates after synthesis. In addition, the current trend is to move synthesis toward higher levels of abstraction [7]. Therefore, not being able to handle higher-level code will become a serious limitation of such gate-level techniques in the future.

In this work we develop techniques and methodologies to utilize abundant external don't-cares that exist in an SoC design for IP optimization. Our techniques focus on optimizing high-level code directly; therefore, they can scale to larger designs and the optimizations are easier to be verified by designers. In addition, working on the higher-level code can detect many optimization opportunities that are difficult to be identified at the gate level. Our first contribution is a new algorithm that utilizes high-level symbolic simulation to perform formal code-statement reachability analysis and then use the reachability report to identify and remove redundant design code in order to produce smaller netlists after synthesis. Our approach is considerably different from traditional reachability analysis in that our target is design code instead of design states, and it is different from code-coverage analysis because we can formally prove code reachability. Our second contribution is an innovative synthesis construct called *sym_wait* that can accelerate symbolic simulation. This construct first verifies the latency of different symbolic traces and then merges them into one trace, thus improving the runtime of symbolic simulation and reducing its memory usage. We utilized our reachability analysis techniques in our third contribution, a methodology that uses existing verification environments or surrounding blocks for circuit optimization. This methodology reuses constrained-random testbenches prevalent in circuit verification to model the surrounding environment of the design under optimization, and then it performs circuit optimization by exploiting the don't-cares that exist in the testbench. Since different input constraints can produce different circuit optimization opportunities, engineers can use different testbench constraints to model different software requirements, and then apply our techniques to perform hardware/software co-optimization and co-exploration to find the best trade-off between software design and hardware complexity. Note that the testbench can be under-constrained and thus is often easy to develop. In the extreme case, even if the inputs are completely random, we can still identify dead code in the design and reduce design size.

As our empirical evaluation in Section V shows, different sets of instructions exhibit different optimization opportunities and can produce different sizes of optimized netlists. For example, when 16 types of instructions are allowed in DLX, only 3.3% cell count reduction can be achieved. But if the software only uses 8 types of instructions, 14.7% reduction can be achieved. These results suggest that our solutions can optimize IPs in an SoC effectively.

The rest of this paper is organized as follows. In Section II we briefly review existing verification methods and provide necessary background. Techniques to perform synthesis optimizations using high-level symbolic simulation are presented in Section III. In Section IV we describe several insights gained during the implementation of our methodologies and outline how to ensure the correctness of the optimized circuits. Empirical results are shown in Section V, and Section VI concludes this paper.

II. Related Work

In this section we first summarize the benefits and limitations of recent verification techniques, and then briefly describe the characteristics of symbolic simulation. Finally, several techniques for computing and utilizing don't-cares are presented.

A. Prevalent Verification Techniques

Verification techniques for hardware designs have been extensively investigated over the past decades [1]. Among the verification methods that exist today, logic simulation using pre-defined or constrained-random patterns is the most commonly used. However, there are several known limitations such as incomplete corner-case coverage. To overcome the limitations of simulation-based verification, formal verification techniques have been developed, such as model checking and reachability analysis [9, 17]. Formal verification tools exploit mathematical methods and offer more comprehensive verification compared to simulation-based methods. However, formal tools often have scalability issues; in addition, writing assertions and constraints to describe

978-1-4244-5766-3/10/$26.00 ©2010 IEEE

the intended behavior of the design can be challenging. Therefore, simulation-based techniques still remain prevalent nowadays. One major advantage of our techniques is that we can reuse the testbenches developed for simulation-based verification for circuit optimization, thus eliminating the need to write new code to model the environment.

B. Symbolic Simulation

Symbolic simulation is a formal verification method that is easier to use than other formal methods since it can work with traditional simulation-based methodologies. The difference between logic and symbolic simulation is that logic simulation only simulates scalar inputs, while symbolic simulation allows the use of Boolean variables, or symbols, as input values. Therefore, symbolic simulation produces Boolean expressions (also called symbolic traces) instead of scalar values as outputs. Since each symbol represents arbitrary values, symbolic simulation can handle all possible input patterns simultaneously.

While most symbolic simulators are restricted to the gate level, RTL symbolic simulation has become popular recently due to several unique advantages over other methods that we will analyze in detail in Section III-B. One such work is by Kolbl et al. [11, 12]. Their approaches focus on how to handle RTL Verilog constructs containing delay and array structures in a BDD-based symbolic simulator. Since BDD can explode quickly, modern symbolic simulators use other logic representations and call backend solvers when necessary [5]. Sunkari et al.. [16] also proposed a word-level symbolic simulator; they address the issues on how to accurately evaluate events that mutually trigger each other.

C. Don't-Cares in Logic Synthesis

Don't-Cares (DCs) play an important role in the field of logic synthesis because they provide additional flexibility for Boolean reasoning and optimization. DCs can be classified into two types: controllability and observability don't-cares. Controllability Don't-Cares (CDCs) occur when certain values of the subnetwork can never be produced under any legal primary input values, and Observability Don't-Cares (ODCs) occur when the values of subnetwork do not affect any primary output.

Espresso [15] is one of the most well-known Boolean minimization tools which can synthesize and optimize truth tables with DCs. Recently, many studies have been proposed to accurately and efficiently compute don't-care values in synthesis applications [18, 19]. In addition, SAT-based techniques for achieving better scalability and performance have also been developed [13, 22]. As the article by Ranjan et al. [14] pointed out, handling don't-care states through high-level formal verification can provide additional benefits in synthesis and verification. However, most existing work that utilizes DCs focuses on the gate-level, which cannot be applied to high-level synthesis directly. In addition, few studies focus on how external DCs should be encoded and utilized. To address such problems, several techniques have been proposed recently [3, 6].

Note that handling don't-cares at the RTL using logic simulation is often inaccurate due to X-pessimistic and X-optimistic characteristics [2, 21]. Therefore, mismatch between RTL and gate-level netlists could occur. One solution for such a problem is to model the X using a symbol, and then employ symbolic simulation to accurately handle X-propagation at the RTL [6].

III. Circuit Optimization Using Constrained-Random Testbench

The objective of our work is to propose a methodology that utilizes the unique proving power provided by high-level symbolic simulation to automatically identify external don't-cares encoded in existing constrained-random testbenches for logic optimization. In this section, we first formulate the problem and then perform a detailed analysis on why RTL symbolic simulation is more suitable than other formal methods for our methodology. Next, we propose a novel technique which uses symbolic simulation to perform code-statement reachability analysis that can utilize the external don't-cares implicitly encoded in the testbenches. Finally, we illustrate the whole flow of our methodology.

A. Problem Formulation

The main objective of our synthesis optimization is to reduce the sizes of synthesized netlists by identifying RTL code statements that become redundant due to don't-care conditions encoded in the testbench. This optimization problem is formulated as follows. Given a design containing N conditional code blocks and a constrained-random testbench that can generate all sets of possible input patterns, we seek to remove unused blocks and produce a smaller RTL design based on the given inputs. This objective is guaranteed not to change the design behavior because the removed code is proven to be unreachable under the given inputs. Note that in our formulation, the input patterns can be either properly-constrained or under-constrained according to different requirements as long as all legal inputs can be generated. Even if no testbenches exist, we can still utilize part of the block that feeds into the block under optimization as the testbench, as we will show in Section III-D.

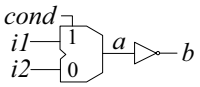
B. Analytical Study of Symbolic Simulation

In this subsection we perform an analytical study of high-level symbolic simulation and explain why it is more suitable for our optimization methodology than other formal methods. The most prominent difference between high-level symbolic simulation and other formal methods is that it is intrinsically a combination of software and hardware verification techniques. In particular, it is similar to hardware-based methods in that it generates Boolean expressions to describe the functions between design variables. Therefore, it is easy to maintain functional correctness of the hardware design during our optimization process. On the other hand, it also traces the execution of code statements like software verification methods, thus allowing the utilization of don't-cares at the RTL. Take Figure 1(a) for example, hardware methods convert the code to a gate-level netlist as shown in Figure 1(b) for analysis. While the netlist faithfully captures the logic relationship among variables a, b, $i1$ and $i2$, it is more difficult to handle the $\#1$ delay that may exist in testbenches and the $display message that can be useful for debugging. Although software symbolic simulation is able to execute the $display statement, it has trouble handling the delay and the event-trigger constructs due to the sequential nature of software program execution. Symbolic simulation combines the best of both worlds so it can generate precise Boolean expressions as shown in Figure 1(c), as well as show the $display messages when symbolic simulation executes those code statements ($S3$ and $S5$). Note that in Figure 1, we use subscript to represent the time that a symbol is generated, and we assume the execution of $S1$ is at time 0. Subscript "s" denotes the initial value of the variable.

High-level symbolic simulation and other gate-level based formal methods have many differences. It is noteworthy to mention that reachability analysis in gate-level techniques typically operates on complex state transition diagrams. Synthesis optimization algorithms then identify unreachable states and change state encoding to reduce the number of logic gates [10, 20]. One limitation of gate-level state optimization is that it is difficult to map the optimizations back to the RTL for designers' review when states are not preserved, making verification more difficult. On the other hand, high-level symbolic simulation can perform code statement reachability analysis in designs/testbenches. Therefore, it is much simpler for designers to trace the changes.

(a)
```
always @(cond, i1, i2) begin
S1:   if (cond) begin
S2:     a = i1;
S3:     $display("Important debugging message");
      end else
S4:     #1 a = i2;
S5:   $display("Execution finished"); end
S6: always @(a) b =~ a;
```

(b)



(c)

a_0 = casecond($(cond_{@0})$ – 0: $a_{@s}$, 1: $i1_{@0}$)
b_0 = casecond($(a_{@s} == a_{@0})$ – 0: $\sim a_{@0}$, 1: $b_{@s}$)
a_1 = casecond($(cond_{@0})$ – 0: $i2_{@1}$, 1: $i1_{@0}$)
b_1 = casecond($(a_{@0} == a_{@1})$ – 0: $\sim b_{@0}$, 1: $\sim a_{@1}$)

Fig. 1. Example of high-level symbolic simulation: (a) RTL code; (b) gate-level netlist; (c) symbolic trace of variables a and b, where $a_{@s}$ and $b_{@s}$ denote initial values of variables a and b, and $a_{@1}$ is the value of a at time 1.

C. Code Statement Reachability Analysis

Code statement reachability analysis is a technique that performs an exhaustive exploration of reachable code statements. Unlike state-based reachability analysis commonly used in hardware verification and synthesis, high-level symbolic simulation provides capability to perform software-like statement reachability analysis. Specifically, each conditional code block in high-level symbolic simulation is treated as a branch which involves one or more statements that will always be executed under the same condition. To identify whether a conditional code block can be reached or not, we utilize the formal nature of symbolic simulation to evaluate all possible values of variables involved in the conditions to execute those statements. The advantage of utilizing reachability analysis at the RTL for design optimization is twofold. (1) A few lines of RTL code can be synthesized into numerous gates. By identifying unreachable code statements at the RTL and removing them, we can potentially remove thousands of gates simultaneously. (2) Unlike traditional hardware optimization techniques that utilize unreachable states to optimize sequential elements only, our code-statement reachability analysis can also identify redundant code statements that will be mapped into combinational logic.

Although the exhaustive nature of symbolic simulation allows us to explore all the paths to reach each conditional block, keeping track of all the symbolic conditions to enter the block is still a non-trivial task. One major reason is that hardware design languages include semantics to model hardware behavior like events and delays. These semantics allow the execution of a code segment to be triggered under numerous reasons by some code that may seem to be unrelated to the code segment itself. To handle these semantics, we modify the event-driven symbolic simulation algorithm to make sure we can correctly keep track of the symbolic conditions when entering each conditional block. The modified algorithm for symbolic code-statement reachability analysis is shown in Figure 2. In the algorithm, an event can be a delay or a signal change condition that triggers the execution of certain code blocks. Variable *curr_sym_cond* saves the symbolic condition when executing the code statement. The variable is updated when entering or leaving a conditional block, as shown in lines 4-8. If a new event needs to be generated, we save the current symbolic condition to the event's sym_cond field. In this way, all the code segments triggered by the event will have the correct symbolic condition.

To perform statement reachability analysis, we use SAT solvers to check whether *curr_sym_cond* in line 5 is satisfiable or not. This is different from other work on symbolic simulation [11] which uses BDDs to encode all symbolic conditions thus no SAT calling is necessary. If *curr_sym_cond* is satisfiable, we mark the conditional block as reachable. After simulating all the required cycles, if a conditional block is never reachable, then we report the block as unreachable. Note that

```
1   event = event_queue.pop();
2   curr_sym_cond = event.sym_cond;
3   while execute statement triggered by event
4     if statement is a conditional block with condition cond
5       curr_sym_cond &= cond;
6       do not execute statment if curr_sym_cond is proven to be 0;
7     else if leaving conditional block with condition cond
8       restore curr_sym_cond by removing cond as constraint;
9     else if a new event nevent needs to be generated
10      nevent→sym_cond = curr_sym_cond;
11      event_queue.add(nevent);
12    statement = statement.next;
```

Fig. 2. Pseudo code of symbolic code statement reachability analysis.

in symbolic simulation, if the symbolic condition is unsatisfiable, then we can prune the simulation by not executing the conditional block, thus reducing overall runtime and memory use. As a result, performing reachability analysis may not decrease the performance of symbolic simulation significantly.

Sometimes designers use Xs to explicitly tell the synthesis tools to treat certain input combinations as don't-cares so that the code can be optimized. For example, it is common to see code that assigns Xs to a variable in the default branch of a select/case statement. To support this usage in our reachability analysis, whenever an X is assigned to a variable, we replace the X with a symbol. In this way, if the X does affect the reachability of a conditional block, the symbolic condition to enter the block will become satisfiable and the block will be marked reachable. The ability to correctly handle Xs is one reason why the reachability analysis technique described earlier cannot be replaced by traditional code-coverage analysis performed using logic simulation: due to X optimism and pessimism, logic simulation cannot correctly handle such Xs and may produce incorrect reachability reports.

D. Overall Flow of Our Methodology

Since constrained-random testbenches are often readily available for most designs, we reuse them for reachability analysis and synthesis optimizations. The flow of our methodology is shown in Figure 3. First, for those variables which are assigned random values in the constrained-random testbench (e.g., $random in Verilog), replace $random with symbols. Next, we perform code statement reachability analysis for T cycles. In this way, symbols are propagated to the design, and symbolic traces for conditional statements can be produced under the constraints. After performing reachability analysis using techniques described in Section III-C, a report is produced for code reachability, and each code statement is identified as either reachable or unreachable. Since symbolic simulation can evaluate all possible (constrained) values for inputs, unreachable code statements are seen as redundant and can be removed from the design. Finally, we execute traditional logic synthesis tools to produce the optimized netlist. In our methodology, CDCs in the symbolic conditions created by the constrained-random testbench are utilized for design optimization.

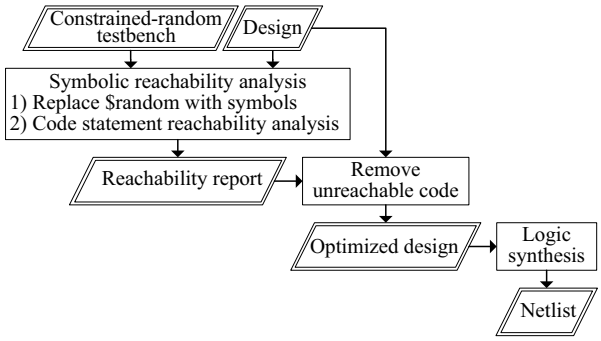


Fig. 3. Flow of synthesis optimization using symbolic code statement reachability analysis.

Even if constrained-random testbenches do not exist, our technique can still be used for circuit optimization in an SoC environment, and it works as follows. Suppose that the inputs of block B come from block A, we can extract all or part of the logic in block A that fans out to block B. Next, we apply pure random inputs to the extracted logic and use it as a testbench to optimize block B. In this way, we can optimize IP cores in SoC designs even if constrained testbenches do not exist. As Chayut [4] points out, this methodology is desirable in SoC designs because it allows synthesis optimizations to work across block boundaries.

Symbolic simulation can evaluate all possible values simultaneously; however, it is still bounded in nature. Hence, additional conditions need to be considered to ensure the correctness of the optimized circuit. More discussions on how to verify the optimized design are provided in Section IV-C.

IV. Implementation Insights and Optimization Correctness

For large designs, symbolic traces can be very complex and formal analysis can be challenging. In particular, events and delays that are common in testbenches and RTL designs can create numerous symbolic traces, all under different conditions or delays, making symbolic simulation inefficient. To solve this problem, we propose two refinements in this section to improve the performance of symbolic reachability analysis. We then describe how to ensure the correctness of the designs optimized by our methodology.

A. Combining Logic and Symbolic Simulation for Reachability Analysis

Logic simulation has been widely used in code coverage analysis. With random inputs, logic simulation can efficiently determine the reachability for the conditional blocks. To leverage the strengths of both logic and symbolic simulation, we first perform logic simulation for a period of time to identify the conditional blocks that are reachable. Typically, logic simulation is performed until code coverage saturates. Next, we use symbolic simulation to check all conditional blocks that are still not reachable. Since logic simulation is extremely fast, this method can greatly reduce the use of formal analysis and achieve better performance. Note that formal analysis is still necessary to ensure the correctness of reachability analysis since logic simulation may not hit all possible reachable code statements due to its random nature.

B. Merging Symbolic Traces

Like all formal methods, the comprehensive nature of symbolic simulation also has its drawbacks. Most notably, it has to keep track of code execution under all possible conditions. For instance, it has to execute the "if" statement in Figure 1(a) twice, one with "$cond$ = true" and one with "$cond$ = false", generating two different simulation traces. If no delay exists in either branch, then the trace explosion problem can be alleviated by merging both traces after executing the "if" statement to produce a Boolean expression like "$a = cond?i1 : i2$" for the execution of future code starting at $S5$. However, when delays exist, merging becomes unlikely and the execution traces have to be split afterwards. Such splitting can produce a large number of traces, all under different conditions or delays, making symbolic simulation inefficient. Since delays appear frequently in RTL designs or testbenches, such a phenomenon may become the bottleneck of our optimization.

To solve this problem, we propose a new construct called *sym_wait* to reduce the complexity of symbolic simulation. The construct can be inserted by the designer to appropriate code locations to merge different traces and greatly simplify symbolic simulation. When *sym_wait* is executed by symbolic simulation, it keeps track of all the symbolic traces and verifies whether or not all the traces reach *sym_wait* within a time period. If not, then *sym_wait* flags an error to indicate that it cannot be used to merge the traces. Otherwise, it merges the symbolic traces into one and continues symbolic simulation using the merged trace. The reason why we verify the latency of traces first before merging them is to make sure all the traces can finish in the current cycle or transaction so that we can merge them. Otherwise, reachability analysis may be incorrect.

A more detailed description of *sym_wait* is shown in Figure 4. It determines whether all code execution paths, all under different conditions, can reach the specified code statement within a timeout period (denote as T_o). Specifically, the time difference between the first and the last simulation paths that reach *sym_wait* should not be larger than T_o. Once this property holds, the execution traces can be merged, creating only one trace that should go forward as if the delay does not exist. Otherwise, the verification construct is violated. The time-out limitation used here provides additional checks for latency-related problems in testbenches or designs. More specifically, this construct verifies that the latency of all traces is within an acceptable range before merging them to ensure the correctness of the merging.

```
1  Construct sym_wait(T_o)
2    if(all path sequences from reset state under
3       all conditions can reach this statement within T_o)
4       merge different Boolean expressions at this point;
5    else
6       generate a illegal path sequence that reproduces the problem;
```

Fig. 4. Syntax and semantics of *sym_wait*.

Figure 5 is a simple illustration to show how *sym_wait* works on the design described in Figure 1. Suppose that *sym_wait* is added before statement $S5$ and timeout is 0. When *sym_wait* is first reached by one of the branches ($S5$, $cond$=true), it will block the execution until all possible branches reach this statement or timeout occurs. In this case, the other branch reaches $S5$ at time 1, thus violation occurs. Similarly, suppose that as long as statement $S5$ can be reached before time 1 under all conditions (e.g., $cond$ is constrained to 1), the property of timeout is held, and only one trace will go forward. As shown in our empirical results, this technique can significantly reduce the complexity of symbolic simulation.

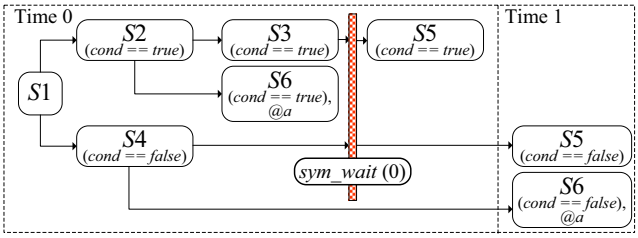


Fig. 5. Statement execution diagrams to illustrate how *sym_wait* checks the time-out of statement reachability.

C. Ensuring the Correctness of Optimizations

Although symbolic simulation can evaluate all possible values simultaneously, it can only ensure the correctness and the completeness of verification within the simulated time period. One way to solve this problem is to use "proof by induction" that is described in detail in [8]. The basic idea behind this method is that if the state in the last simulated cycle is a subset of any state before the last cycle, then the properties verified to hold for the simulated cycles will hold forever. In reachability analysis, if the condition mentioned above is satisfied, then all the unreachable code statements are guaranteed to be unreachable. By performing state reachability analysis, we can determine the number of cycles that should be performed in order to satisfy the condition that allows "proof by induction" to work.

TABLE I
CHARACTERISTICS OF BENCHMARKS.

Design	Description	#Cond. blocks	#Cells	Timing slack
DLX	5-stage pipeline CPU, MIPS architecture	274	13902	2659 ps
Alpha	5-stage pipeline CPU, 64-bit registers/instructions/datapaths	175	31381	3269 ps

Sometimes it may be difficult to simulate enough cycles to make the condition required by proof-by-induction hold. To address this problem, one can underconstrain the initial state by making some of the state bits unconstrained. Although under-constraining the design may result in loss of optimization opportunities because unreachable code may be flagged as reachable, this is not an issue since our goal is design optimization, not design verification. Another approach to solve this problem is to generate constraints for the inputs as described in [3]. As long as the inputs comply with the generated constraints, design correctness can also be guaranteed.

V. EXPERIMENTAL RESULTS

In this section, we first measure the effect of *sym_wait* using a Crossbar switch design, and then perform optimization on several benchmarks, including a DLX processor and an Alpha processor. The characteristics of these benchmarks are listed in Table I. Our experiments were performed using a commercial symbolic simulator called Insight [24] running on a Linux workstation with 2 GHz Quad-Core Xeon processors and 40 GByte main memory. The SAT engine we used is ABC [23] compiled in 64-bit mode. A state-of-the-art commercial logic synthesizer is used to synthesize the original RTL and the code optimized by our techniques. As a result, the experimental results can faithfully show what designers will benefit from our methods in their design flows.

A. Case study: Crossbar Switch

The purpose of this case study is to show how *sym_wait* can improve symbolic simulation analysis. Crossbar switches can be found in many designs including routers, network-on-chip circuitry and communication chips. Our switch contains two input ports and two output ports. It can forward a packet from any input to any output based on its priority bit and a round-robin arbitration scheme. Note that in the testbench that generates data packets for verification, many execution branches will be created in symbolic simulation because data length can be random, and this behavior makes symbolic simulation less efficient.

We prepared a testbench that connects a driver and a receiver through a FIFO and used this configuration to evaluate the effectiveness of *sym_wait*. As we have shown in Section IV-B, numerous simulation traces have to be produced to track all the packets due to different payload sizes that require different numbers of cycles to transmit. However, *sym_wait* can merge those traces again. In this case study, we found that when *sym_wait* is applied, runtime reduced from 1525 seconds to 442 seconds, achieving 71% reduction. Furthermore, memory consumption reduced by 44% due to branch merging.

B. Cast Study: DLX Processor

In this case study, we developed five testbenches to customize the DLX implementation provided by the BugUnder Ground (BUG) project from Michigan [25]. For each testbench, only certain specified instructions are allowed. In this way, the logic for the unused instruction set can be removed to reduce the size and power consumption of the new design. To ensure the correctness of customized design, DLX was initialized to the state in which all registers were symbols, and it was simulated symbolically for 14 cycles. Since all possible states can be reached in 7 cycles under this configuration, it is guaranteed that the state in the 14th cycle is a subset of the state at the 7th cycle; thus ensuring the correctness of the optimized design.

In the ALU control block, for synthesis optimization, the designer assigned X to the default branch of the case statement that selects the ALU operation. Since some instructions, like BEQ, does not match any case condition, the control signal of the ALU may have any value for such instructions depending how the synthesis tool optimizes the case statement. Our reachability analysis correctly handled this X situation and showed that all ALU operations are necessary, while logic simulation only matches the default branch. This observation shows that symbolic reachability analysis can produce much more accurate results than logic simulation. Since the output of ALU will not be used by the BEQ instruction, this is not a design bug. However, our analysis shows that unpredictable ALU operation may be selected by the BEQ instruction, and this behavior may consume unnecessary power. In our experiments we removed this design flaw before synthesis.

The results after optimization are shown in Table II. When fewer numbers of instruction types are allowed, more code blocks become unreachable, and more optimization can be achieved. Note that most storage devices are always reachable in this design; therefore, even when only the NOP operation is allowed, there are still 122 reachable blocks. In this case study, the maximum reduction of gate count is 81.4% and the minimum one is 3.3%. In addition, the total timing slack is also reduced due our optimizations, leading to a better performance and lower power consumption. It is observed that runtime of these testbenches does not have an apparent trend. The reason is that when more instructions are allowed, more code blocks will become reachable. Since satisfiable problems are often faster to solve than unsatisfiable problems, SAT solvers can spend less time solving the symbolic conditions. However, more reachable blocks also mean more symbolic simulation must be done because more conditional blocks will be entered. Therefore, the runtime does not show an obvious trend.

C. Case Study: Alpha Processor

In this section, we customize an Alpha design using symbolic reachability analysis. Similar to the previous case study, we use the Alpha implementation by the BugUnder Ground (BUG) project from Michigan [25]. We prepared six testbenches to constrain the instructions allowed to be used by the Alpha processor and the results are presented in Table III. It is observed that if only instruction NOP is allowed, a lot of code blocks can be removed and the produced netlist is much smaller. As the number of instruction types increases, more code blocks can be reached, thus the gate-count reduction ratio is decreased. It is interesting to note that after optimizing the last testbench that has 17 instructions, there was slight increase in gate count. The reason is that removing code may not always reduce synthesized gate count if synthesis tools do not handle don't-cares that became available in the new code wisely. But the general trend shows that we can indeed simplify the design. In addition to gate-count reduction, we also observed that timing slack was also reduced, which shows that our methodology can improve both the size and the performance of SoC designs. These results also show that one can find a trade-off between software and hardware for the best system performance.

To better illustrate this idea, we also tried a different set of instructions that does not include MULQ and the results show that 34.7% gate-count reduction can be achieved. Therefore, if MULQ is never or rarely used in the program that will be executed on the Alpha processor, engineers can remove this instruction from the supported instruction list. However, if multiplication is used often in the program, then

TABLE II
DLX OPTIMIZATION THROUGH CODE STATEMENT REACHABILITY UNDER DIFFERENT COMBINATIONS OF INSTRUCTIONS.

Instruction allowed (DLX)	Run time	#Cond. blocks	#Cells	Reduction ratio	Timing slack
NOP	1 sec	122	2426	81.4%	1248 ps
ADD, ADDI, NOP	100 min	148	7793	68%	1563 ps
ADD, ADDI, SW, LW, NOP	103 min	165	9240	29.4%	1606 ps
ADD, ADDI, SW, LW, SRL, SLL, SRA, BEQ, NOP	97 min	176	11170	14.7%	2306 ps
ADD, ADDI, AND, ANDI, XOR, SLT, SLTI, SW, LW, SRL, SLL, SRA, BEQ, BNE, J, JAL, NOP	138 min	208	12661	3.3%	2596 ps

TABLE III
ALPHA OPTIMIZATION THROUGH CODE STATEMENT REACHABILITY UNDER DIFFERENT COMBINATIONS OF INSTRUCTIONS.

Instruction allowed (Alpha)	Run time	#Cond. blocks	#Cells	Reduction ratio	Timing slack
NOP	1 sec	98	1339	95.7%	747 ps
ADDQ, MULQ, CMPEQ, NOP	25 min	120	27941	10.9%	3240 ps
ADDQ, MULQ, CMPEQ, LDQ, STQ, NOP	24.5 min	127	28300	9.8%	3280 ps
ADDQ, MULQ, CMPEQ, LDQ, STQ, JMP, BSR, SRL, SLL, SRA, NOP	18.5 min	142	30265	3.7%	3268 ps
ADDQ, SUBQ, MULQ, CMPEQ, CMPULE, LDQ, STQ, JMP, RET, BSR, SRL, SLL, SRA, AND, BIS, XOR, NOP	15.5 min	149	32195	-2.6%	3298 ps
ADDQ, SUBQ, CMPEQ, CMPULE, LDQ, STQ, JMP, RET, BSR, SRL, SLL, SRA, AND, BIS, XOR, NOP	15.5 min	146	20476	34.7%	1848 ps

this instruction must be preserved. This example shows how engineers can use our techniques to perform hardware/software co-optimization and co-exploration.

VI. CONCLUSIONS

In this work, we proposed techniques and methodologies that utilize abundant external don't-cares which exist in a System-on-Chip (SoC) design for optimization. More specifically, we utilize existing testbench or other design blocks to search for such don't-cares and perform symbolic code-statement reachability analysis to identify code blocks that become redundant due to the don't-cares. Since our goal is design optimization instead of verification, the testbenches can be under-constrained and thus is easy to develop. Furthermore, our approach is guaranteed not to change design behavior because the removed code will never be executed under the inputs constraints. Because our approach focuses on high-level code directly, it can scale to larger designs and can identify many optimization opportunities that are difficult to be detected at the gate level. Our empirical results using DLX and Alpha show that our symbolic reachability analysis can optimize designs in SoC environments, both in terms of gate count and timing slack. In addition, our methodology allows hardware/software co-optimization and co-exploration to find the best trade-off between hardware and software complexity.

ACKNOWLEDGMENT

This research was supported by the National Science Council, Taiwan under Grant NSC 97-224-E-002-216-MY3; Excellent Research Projects of National Taiwan University,95R0062-AE00-05; and Small Business Innovation Project of Ministry of Economic Affairs, Taiwan, 1Z960401.

REFERENCES

[1] J. Bhadra, M. Abadir, L.-C. Wang, and S. Ray, "A Survey of Hybrid Techniques for Functional Verification," *IEEE Design and Test of Computers*, Vol. 24(2), pp 112-122, 2007.
[2] D. Brand, R. A. Bergamaschi and L. Stok, "Be Careful with Don't Cares," *DAC'95*, pp. 83-86.
[3] K.-H Chang, V. Bertacco, and I. L. Markov, "Customizing IP Cores for System-on-Chip Designs Using Extensive External Don't-Cares," *DATE'09*, pp. 582-585.
[4] I. Chayut, "Next-Generation Multimedia Designs: Verification Needs", *DAC'06*, Section 23.2.
[5] H. Z. Chou, I. H. Lin, C. S. Yang, K. H. Chang and S. Y. Kuo, "Enhancing Bug Hunting Using High-Level Symbolic Simulation", *GLSVLSI'09*, pp.417-420.
[6] H. Z. Chou, K. H. Chang, and S. Y. Kuo, "Handling Don't-Care Conditions in High-Level Synthesis and Application for Reducing Initialized Registers," in *DAC'09*, pp. 412-415.
[7] P. Coussy and A. Morawiec, "High-Level Synthesis: from Algorithm to Digital Circuit," Springer, 2008.
[8] M. K. Ganai and A. Gupta, "SAT-based Scalable Formal Verification Solutions," Springer, 2007.
[9] M. K. Ganai and A. Aziz, "Improved SAT-based Bounded Reachability Analysis," *ASPDAC'02*, pp. 729-734.
[10] T. Kam, T. Villay, R. Brayton and A. Sangiovanni-Vincentelli, "A Fully Implicit Algorithm for Exact State Minimization," *DAC'94*, pp. 684-690.
[11] A. Kolbl, J. Kukula and R. Damiano, "Symbolic RTL Simulation," *DAC'01*, pp. 47-52.
[12] A. Kolbl, J. Kukula, K. Antreich and R. Damiano, "Handling Special Constructs in Symbolic Simulation," *DAC'02*, pp. 105-110.
[13] A. Mishchenko and R. K. Brayton, "SAT-Based Complete Don't-Care Computation for Network Optimization," *DATE'05*, pp. 412-417.
[14] R. Ranjan, Y. Antonioli, A. Hunter, and O. Petlin, "Formal Verification Enables Safe X Handling," Dec. 2008. (available at http://www.scdsource.com/article.php?id=324
[15] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," *IEEE TCAD*, Vol. 6(5), pp. 727-750, 1987.
[16] S. Sunkari, S. Chakraborty, V. Vedula and K. Maneparambil, "A Scalable Symbolic Simulator for Verilog RTL," *workshop on IEEE MTV*, pp. 51-59, 2007.
[17] S. Safarpour, A. Veneris and R. Drechsler, "Improved SAT-based Reachability Analysis with Observability Don't Cares," *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 5, pp. 1-25, 2008.
[18] H. Savoj and R. K. Brayton, "Observability Relations and Observability Don't Cares," *ICCAD'91*, pp. 518-521.
[19] N. Saluja and S. Khatri, "A Robust Algorithm for Approximate Compatible Observability Don't Care (CODC) Computation," *DAC'04*, pp. 422-427.
[20] E. M. Sentovich, H. Toma, and G. Bérry, "Latch Optimization in Circuits Generated from High-level Descriptions," *ICCAD'97*, pp. 428-435.
[21] M. Turpin, "The Dangers of Living with an X," SNUG Boston, 2003.
[22] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't-Cares," *DAC'06*, pp. 229-234.
[23] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70930. http://www.eecs.berkeley.edu/∼alanmi/abc/
[24] Avery Design Systems Inc., http://www.avery-design.com
[25] Bug UnderGround, http://bug.eecs.umich.edu