Logic Synthesis and Circuit Customization Using Extensive External Don't-Cares

Kai-hui Chang[*♯], Valeria Bertacco[*], Igor L. Markov[*†], Alan Mishchenko[‡]
[*]EECS Department, University of Michigan, Ann Arbor, MI
[♯]Avery Design Systems, Andover, MA
[†]Synopsys, Inc., Sunnyvale, CA
[‡]EECS Department, University of California, Berkeley, CA

Traditional digital circuit synthesis flows start from an HDL behavioral definition and assume that circuit functions are almost completely defined, making don't-care conditions rare. However, recent design methodologies do not always satisfy these assumptions. For instance, third-party IP blocks used in a system-on-chip are often over-designed for the requirements at hand. By focusing only on the input combinations occurring in a specific application, one could resynthesize the system to greatly reduce its area and power consumption. Therefore we extend modern digital synthesis with a novel technique, called SWEDE, that makes use of extensive external don't-cares. In addition, we utilize such don't-cares present implicitly in existing simulation-based verification environments for circuit customization. Experiments indicate that SWEDE scales to large ICs with half-million input vectors and handles practical cases well.

Categories and Subject Descriptors: B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Circuit customization, don't-care optimization, logic synthesis

1. INTRODUCTION

Due to the increasing demand for integrated circuits to provide more functions while consuming less power, designing a new chip becomes more and more difficult. One way to reduce this design effort is to reuse previously designed circuits, such as Intellectual Property (IP) blocks and general-purpose processors. This approach, however, may result in designs with unnecessarily large area and power consumption because they are over-provisioned with respect to the target functionality. On the positive side, system performance and cost may be improved by customizing reused components to the target applications and environment. This novel optimization, which we call *design specialization*, poses a new synthesis challenge, which differs from traditional formulations by the abundance of external don't-cares. In fact, both academic and commercial synthesis tools available today appear to be

Author's address: Kai-hui Chang[*♯], Valeria Bertacco[*], Igor L. Markov[*†], Alan Mishchenko[‡], [*]EECS Department, University of Michigan, Ann Arbor, MI, [♯]Avery Design Systems, Andover, MA, [†]Synopsys, Inc., Sunnyvale, CA, [‡]EECS Department, University of California, Berkeley, CA. Email {changkh,valeria,imarkov}@eecs.umich.edu, alanmi@eecs.berkeley.edu

structured and optimized to extract optimization opportunities from small, localized sets of external don't-cares. While this approach succeeds on mainstream synthesis instances, it does not perform well on the type of instances generated in the context of design specialization and cannot handle some cases at all. Our experimental study revealed that the performance of existing tools, such as Espresso [Rudell and Sangiovanni-Vincentelli 1987] and some commercial synthesis tools, greatly deteriorates when extensive don't-cares are added. In addition, several other tools, such as ABC [ABC 2007] and commercial tools, do not handle our problem instances at all, or do not provide specification formats for this situation. The latter problem is especially serious because without a simple and efficient way to represent don't-cares for synthesis tools, the adoption of circuit-customization methodologies will be much more difficult.

In this work we address two types of synthesis problems in the presence of extensive external don't-care sets. The first type assumes that the care-terms are known and represented using a truth table while the circuit structure is unknown. To solve this problem, we propose CleanSlate and InterSynth algorithms that synthesize the truth table from scratch. The second problem type assumes that an initial circuit already exists for customization. To solve this problem, we developed a FastShrink algorithm, which takes as input an optimized design and *reduces* it based on the specified don't-care set. Note that FastShrink might find optimization opportunities even when applied after CleanSlate. Our approach is based on an important insight: extensive don't-cares allow simple greedy algorithms to quickly produce a reasonably small netlist, and the missed optimization opportunities can be recovered afterward using more sophisticated synthesis techniques. Since this latter step does not consider don't-cares, it can run much faster and leverages existing tools. This two-step process eliminates the need for a time-consuming don't-care optimizer, yet it is still capable of generating high-quality netlists. Our second contribution is a methodology that automatically extracts don't-care information from existing verification environments, which can be either a direct test or a constrained-random testbench, for circuit customization. In this way, designers do not need to encode don't-cares explicitly, which is often difficult and time-consuming. We integrated these techniques in our tool, called *SWEDE* (Synthesis Within an Extensive Don't-care Environment) [Chang et al. 2009]. In our experiments we performed synthesis from truth tables with large don't-care sets and observed SWEDE completing ten times faster than state-of-the-art synthesis tools while producing comparable or smaller circuits. We have also used SWEDE to customize circuits with up to 30K gates and half-a-million input vectors in under two hours on a single processor in most cases.

SWEDE's high performance enables several new synthesis applications and enhances many others, including (1) input constraint synthesis for emulation; (2) acceleration of the most-frequent computation in a unit [Austin et al. 2005; Lakshminarayana et al. 2001; Verma et al. 2008]; (3) customization of third-party IP components in an System-on-Chip (SoC); and (4) support for graceful wear-out of electronic devices [Wagner et al. 2006]. Applications in category (1) can solve current engineering problems, while the others provide new system design paradigms. Our techniques may help address a wide range of emerging concerns in IC design, including increasing verification difficulty, unpredictability of manufacturing [Wagner et al. 2006], and lower-power circuits [Lakshminarayana et al. 2001]. Since our simplified circuits provide correct outputs only within the specified care set, stimuli outside this realm may not be viable. While "soft" application domains

such as multimedia can tolerate these situations well, other applications may require an output flag indicating that a given input cannot be processed correctly. To this end, techniques for masking timing errors, such as the work by Choudhury and Mohanram [2009], can be used to generate the flag.

The rest of the paper is organized as follows. In Section 2 we review previous work and provide necessary background. We then describe our new synthesis techniques in Section 3. Our circuit customization flow and proposed applications are given in Section 4. Experimental results are provided in Section 5, and Section 6 concludes this paper.

2. BACKGROUND

In this section we first review relevant previous work. Next, we describe five important concepts: bit-signatures, Craig interpolation, Shannon entropy, simulation and proof by induction. These concepts are used in our synthesis techniques and circuit customization flows.

2.1 Prior Work on Synthesis with Don't-Cares

Much research has been developed in exploiting don't-cares in synthesis optimization. A classic tool implementing some of the most commonly-used techniques is Espresso [Rudell and Sangiovanni-Vincentelli 1987]. Although other more sophisticated synthesis tools exist, such as ABC [ABC 2007] and MVSIS [MVSIS 2005], these focus specially on synthesis problems with a small number of don't-cares. Moreover, their input specification format makes it impractical to describe a large number of don't-cares. For example, a design that could arise in our problem domain may have 50 inputs and as many as one million care terms, leaving more than 10^{15} combinations to be don't-care terms. In order to specify such a complex set of don't-cares, Brayton et al. [2002] proposed the use of an external netlist to encode them. The construction of such a netlist, however, can be challenging.

In addition to synthesizing from a truth table, it is also possible to optimize a design starting from an existing circuit and simplifying it using the don't-cares via resynthesis techniques such as rewiring [Yamashita et al. 1996], node merging [Plaza et al. 2007] and multi-level logic optimizations [Matsunaga and Fujita 1989]. One major challenge in this context is the representation and manipulation of such don't-cares. For instance, Muroga et al. [1989] proposed the concept of *Compatible Sets of Permissible Functions (CSPFs)*, which was used by Savoj and Brayton [1990] to optimize multi-level networks composed of NOR gates. This representation was later improved by Yamashita et al. [2000] and became *Sets of Pairs of Functions to be Distinguished (SPFDs)*. One major drawback in these techniques is that representing the don't-cares is cumbersome and the related data structures are difficult to work with. Traditionally, these don't-cares are represented by BDDs, often exhausting all memory resources even for moderate-size designs. To address this problem, Sinha proposed an efficient representation of SPFDs based on graphs that can be used in logic resynthesis [Yang et al. 2007]. This approach improves the memory profile of SPFDs, but deteriorates the computing time. Recently, Plaza et al. [2007] relied on bit-signatures generated by functional simulation to approximate observability don't-cares for node merging, followed by SAT-based verification. This approach is faster and is more efficient in memory than other solutions. However, external don't-cares were not used in the optimization. Gorjiara and Gajski [2008] proposed a framework to generate customized circuits and showed that those circuits are much more power efficient than the original versions. Their work demonstrated that IP customization can be extremely

useful. Nonetheless, their techniques cannot customize generic existing circuits like we do. In stead of utilizing don't-cares for circuit optimization, techniques based on logic decomposition and refactoring can also effectively reduce the size of a circuit. To this end, the greedy algorithm proposed by Rajski and Vasudevamurthy [1992] is used in our CleanSlate synthesis flow.

2.2 Craig Interpolation

The concept of Craig interpolation originated in mathematical logic in 1957 and has recently become popular in formal verification. In contrast, we are going to use it in logic synthesis.

DEFINITION 1. *Consider a pair of Boolean functions, $A(x,y)$ and $B(y,z)$, such that $A(x,y) \wedge B(y,z) = 0$, where x and z are variables appearing only in A and B, respectively, and y are common variables of A and B. An interpolant of $A(x,y)$ w.r.t. $B(y,z)$ is a Boolean function I over the common variables y that satisfies the following conditions: $A(x,y) \Rightarrow I(y)$ and $I(y) \Rightarrow \overline{B(y,z)}$ [Craig 1957].*

Consider an unsatisfiable SAT instance composed of two sets of clauses A and B. In this case, $A(x,y) \wedge B(y,z) = 0$. An interpolant of A can be computed from the proof of unsatisfiability of the SAT instance by the algorithm found in [McMillan 2003] (Definition 2). The resulting interpolant is a single-output multi-level logic network represented as an And-Inverter-Graph (AIG) [ABC 2007]. If $A(x,y)$ is the on-set of a function, $B(y,z)$ is its off-set, and $\overline{A(x,y)} \wedge \overline{B(y,z)}$ is its don't-care set, then $I(y)$ can be seen as an optimized version of $A(x,y)$ where the don't-cares are used in a particular way to optimize representation of I.

Interpolation is used in formal verification to compute an over-approximation of the complete set of reachable states [McMillan 2003]. Interpolation has also been used in area- and delay-driven technology mapping into K-input LUTs [Mishchenko et al. 2007]. When applied to technology mapping, interpolation is used to generate new functions for the node being synthesized.

2.3 Bit-Signatures and Entropy

Our FastShrink synthesis technique is based on bit-signatures generated using simulation, which are defined below. Note that a signature is essentially a signal's partial truth table. If the input vectors are applied exhaustively, then the signature of a signal is its complete truth table.

DEFINITION 2. *Given a wire (signal) w in a circuit, computing function f, and input vectors v_1, v_2 ... v_k, the* signature *of w is the bit-vector $(f(v_1),...,f(v_k))$, where $f(v_i) \in \{0,1\}$ represents the output of f given the input vector v_i.*

The second step of the FastShrink technique (see Section 3.3) exploits short-range optimization opportunities in a circuit. Intuitively, signals with less information are easier to optimize. To quickly identify such signals, we use *Shannon entropy*, which is calculated as follows [Shannon 1948]:

$$E_s = -\frac{\#ones}{k}\log_2(\frac{\#ones}{k}) - \frac{\#zeros}{k}\log_2(\frac{\#zeros}{k}) \quad (1)$$

In the equation, E_s is the entropy of signature s, *#ones* is the number of 1s in the signature, and *#zeros* is the number of 0s in the signature. Variable k is the number of bits in the signature and is also the number of vectors applied to the circuit. A larger E means that the signature contains more information.

2.4 Simulation and Proof by Induction

Simulation is one of the most commonly-used verification methods: input stimuli are applied to a circuit's inputs, and the circuit's outputs are checked against expected results. In *logic simulation*, scalar values are applied to the inputs. For example, feeding 2 and 3 to the inputs of an adder will produce 5 on its output. In *symbolic simulation*, symbols are applied to the inputs, and the outputs are logic expressions [Bertacco 2005]. For example, applying a and b to the inputs of an adder will produce $a+b$ on its output. Since a symbol can represent all possible values simultaneously, symbolic simulation has much larger verification power than logic simulation.

One major limitation of simulation-based verification is that it can only check circuit correctness within the simulated cycles. In other words, it can only verify bounded properties. One way to solve this problem is to use proof by induction [Ganai and Gupta 2007]. The basic idea behind this method is that if the initial states before simulation are a superset of the final states after simulating a certain number of cycles, then the properties that hold throughout simulation are guaranteed to hold unboundedly if the circuit is initialized to one of those initial states.

3. CIRCUIT OPTIMIZATION WITH EXTERNAL DON'T-CARES

In this section we formalize the synthesis problem described earlier and propose three circuit-optimization techniques. One shrinks an existing netlist, while the other two perform synthesis starting from a functional specification (truth table). We then illustrate our techniques by example and provide in-depth analysis of our techniques.

3.1 Problem Formulation

We formulate the circuit-specialization problem as follows. Given a circuit, the complete set of all possible input vectors and their output responses (or, equivalently, a functional specification in the form of a truth table), we seek to produce a small netlist that generates the correct outputs for the given inputs. Our solution considers a combinational flattened circuit and performs the optimization without any structural or other information from the user. On the other hand, if structural information is available in the original netlist, it can be used to improve quality of results.

3.2 Fast Synthesis based on Truth Tables

In this section we introduce two fast synthesis techniques based on truth tables. The first one, called CleanSlate, greedily expands cubes and then performs more sophisticated resynthesis to minimize the size of the netlist. The second one, called InterSynth, is based on interpolation.

3.2.1 *The CleanSlate Technique.* Our specification-based synthesis technique, called CleanSlate, starts from a truth table and produces a technology-mapped netlist. The algorithm is outlined in Figure 1: CleanSlate first greedily expands a cube, one literal at a time, similar to the heuristic used in Espresso (lines 1-3). A cube is subsumed by the ex-

panding cube and is eliminated if its outputs are the same as those of the expanding cube. The expansion stops when the cube overlaps another cube with different outputs. After producing an optimized truth table, CleanSlate generates a two-level netlist (line 4), which is fed to ABC for further optimization. Using ABC, CleanSlate first performs fast logic sharing detection of the netlist [Rajski and Vasudevamurthy 1992], and then converts the netlist to an And-Inverter-Graph (AIG) [ABC 2007]. After that, it expands 2-input ANDs in the AIG to multi-input ANDs to create more opportunities for logic sharing detection, and performs AIG resynthesis to optimize the netlist. The procedure in lines 7-10 is applied several times to achieve better optimization (three times in our implementation). At completion, we apply a technology mapping step to produce the final netlist.

```
flow CleanSlate(TruthTable)
 1  foreach row ∈ TruthTable
 2    expand the cube of row until a different cube is reached;
 3    remove other rows in TruthTable subsumed by row;
 4  convert TruthTable to a two-level netlist;
 5  perform fast logic sharing detection of the netlist with [Rajski and Vasudevamurthy 1992];
 6  repeat N times
 7    transform the network to an AIG by 1-level structural hashing;
 8    expand 2-input ANDs in AIG to multi-input ANDs;
 9    perform fast logic sharing detection using [Rajski and Vasudevamurthy 1992];
10    perform AIG resynthesis (AIG balancing, rewriting and refactoring);
11  return netlist by technology mapping the AIG;
```

Fig. 1. The CleanSlate synthesis flow.

The rationale behind our solution is that the large number of don't-cares enables even a greedy algorithm to generate a reasonably small two-level netlist within a short time. We then bypass a time-consuming two-level optimization process, and instead perform multi-level synthesis. As our experimental results in Section 5 indicate, CleanSlate runs 10X faster than exiting tools, handles more complex circuits, and provides comparable or better synthesis quality.

3.2.2 *The InterSynth Technique.* Another specification-based synthesis technique is InterSynth. It is a heuristic procedure that attempts to minimize the size of multi-level logic implementing a given function. There is no guarantee that it will find the smallest or even a relatively good circuit structure, but for most test cases in practical applications (such as interpolation-based model checking), it was found useful for circuit minimization. This approach is based on computing multi-output interpolants, as shown in the pseudo-code of Figure 2. The computation begins by dividing the input patterns into the on-set and the off-set for each output of the design. Next, the multi-output on-sets and off-sets are converted into AIGs and synthesized to reduce the total number of AIG nodes. After that, an incremental SAT problem is solved for each output, by assuming that the on-set and the off-set of this output are true at the same time. The proof of unsatisfiability of this instance is used to derive the interpolant for the output under consideration. The interpolants for all outputs are then combined into a single AIG, which is synthesized to reduce the total number of AIG nodes. Finally, the AIG is mapped into two-input gates as described in Section 3.2.1.

```
function InterSynth(TruthTable)
1   divide TruthTable into on-set and off-set for each output;
2   synthesize shared AIG F0 for off-sets of all outputs;
3   synthesize shared AIG F1 for on-sets of all outputs;
4   for each pair of outputs, f1 and f0, of AIGs F1 and F0
5     derive proof P of f1 ∧ f0 being unsatisfiable;
6     derive interpolant f from the proof P;
7   create shared AIG F from the set of interpolant AIGs {f};
8   synthesize AIG to minimize the numbers of nodes and levels;
9   return netlist by technology mapping the AIG;
```

Fig. 2. The InterSynth synthesis flow

.

InterSynth differs from [Mishchenko et al. 2007] in that it interpolates all primary outputs of the network rather than one node. For this, we extend the interpolation procedure to work for multi-output unsatisfiability proofs derived by solving several incremental SAT problems. The interface of a SAT solver such as MiniSAT [Eén and Sörensson 2003] allows us to specify assumptions for each incremental SAT run. When the run is proved unsatisfiable, assumptions are lifted and the SAT solver can be reused. The assumptions used in the incremental runs express the condition that the on-set and the off-set are true simultaneously. This condition is, by construction, unsatisfiable for the on-set and the off-set. The resulting interpolant is a multi-output AIG such that the function of each output is contained in the interval defined by the on-set of this function and the complement of the off-set.

3.3 Specializing an Existing Netlist

Given an existing netlist, FastShrink uses a two-step process to produce a specialized new netlist. The first step, called *SignalMerge*, quickly merges signals in an existing circuit that are identical under the given input combinations. The second step, called *ShannonSynth*, performs further optimization using local don't-cares. The algorithm of SignalMerge is shown in Figure 3. It first simulates care-term vectors and then merges signals with identical signatures. This allows SignalMerge to leverage both external and internal satisfiability don't-cares to remove redundant gates. Our implementation selects the signal closest to primary inputs for merging to achieve smaller circuit delay. After the signals are merged, unconnected gates are removed. To expose additional merging opportunities, large cells such as AOI, OAI, etc. are decomposed into smaller gates. After signals in the netlist are merged, the netlist can be technology mapped again.

```
function SignalMerge(Circuit)
1   simulate vectors to generate signatures;
2   foreach signals with identical signatures
3     target ← the signal ∈ signals closest to primary inputs;
4     merge signals to target;
5     remove gates with no fanouts;
```

Fig. 3. The SignalMerge algorithm.

```
function ShannonSynth(Circuit)
1   simulate vectors to generate signatures;
2   compute the entropy of each signature;
3   foreach signal whose signature has 20% smallest entropy
4     extract a subcircuit involving signal as its output;
5     build a truth table using the subcircuits' inputs and outputs;
6     resynthesize the truth table using CleanSlate;
7     if (resynthesized netlist is smaller)
8       replace the subcircuit with the resynthesized netlist;
```

Fig. 4. The ShannonSynth algorithm.

Signal merging can remove redundant logic that generates identical signal functions. ShannonSynth pushes the optimization further by reimplementing subcircuits in smaller structures using don't-cares. To quickly identify subcircuits with high optimization potential, we use Shannon entropy to guide our resynthesis. Intuitively, signatures with low entropy contain less information and should be easier to optimize. In our experience we found that for a random subcircuit-extraction technique to produce the same quality as our entropy-guided approach, 50% more runtime is required.

The ShannonSynth algorithm in Figure 4 first simulates vectors in the care terms to generate a signature for each signal. Next, it computes the entropy of each signature. To make sure its resynthesis attempts are worthwhile, the algorithm only tries subcircuits whose output signatures have small entropy (the bottom 20% of all signatures in our implementation). The key idea in this algorithm is that, instead of trying to resynthesize the netlist in the subcircuit, we build a partial truth table using only the subcircuit's input and output signatures so that we can exploit don't-cares. ShannonSynth then synthesizes the truth table using the CleanSlate algorithm. In this step, however, we use Espresso to replace lines 1-3 of CleanSlate to achieve better resynthesis quality. This is appropriate in local resynthesis because the truth tables are small. After an optimized truth table is generated, ABC is still called for further optimization and technology mapping. If the new resynthesized netlist is smaller than the original one, ShannonSynth replaces it.

The goal of ShannonSynth is to find local optimization opportunities by extracting subcircuits from the design and optimizing them using don't-cares. It can find optimizations that SignalMerge cannot find. However, runtime of ShannonSynth can be considerably longer than SignalMerge. As a result, SignalMerge should always be performed first. ShannonSynth can then be applied whenever there is spare machine or time left to achieve further optimization.

3.4 A Circuit Specialization Example

We now illustrate the FastShrink algorithm on a 3-bit ripple-carry adder. In this example, input A can only assume values 3, 4 or 5; while input B has values 1 or 7. SignalMerge first simulates all possible six input combinations on the given adder to produce 6-bit signatures for all internal signals. The circuit annotated with the signatures is shown in Figure 5(a). SignalMerge then merges signals with identical signatures and removes all the gates that are no longer connected (Figure 5(b)). At this point, only 8 out of the 15 gates are still needed, resulting in a much smaller circuit.

To further optimize the circuit, we invoke ShannonSynth. This extracts a subcircuit

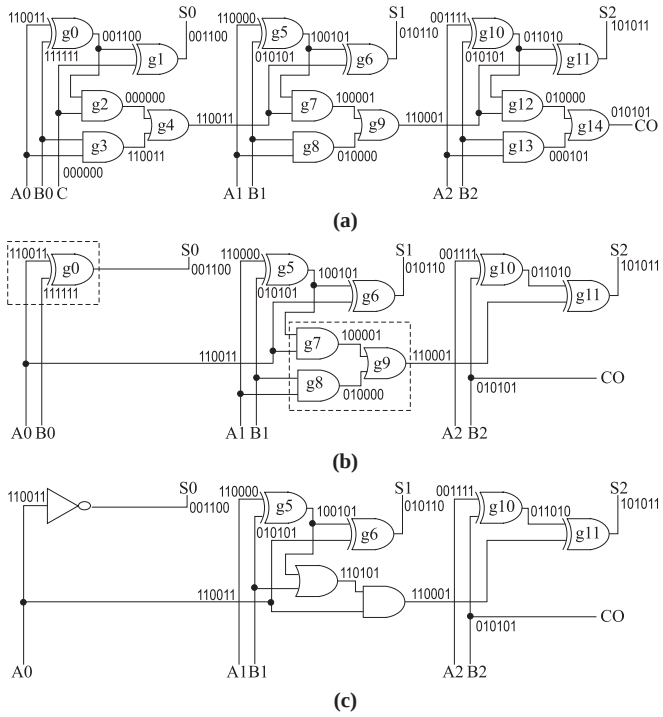

Fig. 5. Ripple-carry adder specialization example: (a) original circuit, (b) after SignalMerge, and (c) after ShannonSynth. Allowed input values are 3, 4, 5 (for A) and 1 and 7 (for B).

composed of gates g7, g8 and g9 to explore further optimizations. First, a truth table is built using the signatures of the subcircuit's inputs and outputs as follows:

A1	A0	B1	g5	g9
1	1	0	1	1
1	1	1	0	1
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1

We then feed the truth table to CleanSlate for synthesis and obtain a new netlist, "g9=A0 & (g5 | B1)", that only uses two gates. Since this resynthesized netlist is smaller, it will replace the original one. Another ShannonSynth run replaces gate g0 with an inverter, and the final result is shown in Figure 5(c). By using the signatures of the subcircuit instead of the netlist for resynthesis, we can fully utilize don't-cares for optimization. This optimization is not performed by many traditional synthesis tools that only use function-preserving netlist transformations. Note that among the 58 don't-care input combinations, 25.9% are still added correctly.

3.5 Analysis

An important property of FastShrink is that every netlist modification it performs always preserves the output responses of the given input vectors. This is because we operate on

signatures, which are simulated values of the input vectors. Since all the changes made by FastShrink preserve signatures, the output responses are also preserved. Moreover, we observe that FastShrink subsumes the common *constant propagation* technique, which is used when a subset of the signals are constant 0 or 1. To simplify our reasoning, we assume that the netlist is decomposed into 1- or 2-input gates, but the same holds in the general case as well.

PROPOSITION 1. *SignalMerge followed by ShannonSynth subsumes the optimizations produced by constant propagation.*

PROOF. Since the output of a 1-input gate can only be constant 0 or 1, SignalMerge connects the output signal to VCC or GND, thus eliminating the gate. Given a 2-input gate, suppose the constant input is the controlling value of the gate, then the output of the gate can only be constant 0 or 1. In this case, SignalMerge proceeds as the 1-input gate. Now suppose that the constant input is not the controlling value of the gate, then the output of the gate can be either identical or the complement to the other input. If the output is identical, then SignalMerge connects it directly to the non-constant input, eliminating the gate. Otherwise, we build a truth table using the gate's input and output signatures and rely on ShannonSynth to simplify the gate to an inverter. □

Finally, note also that a SignalMerge pass guarantees that no two signals are identical in the final circuit, since it merges all the signals with identical signatures.

Our analysis on how current commercial synthesis tools utilize don't-cares suggests that they perform inter-block optimizations by first dissolving the boundaries between the blocks to form a large flattened netlist, and then employing resynthesis techniques such as those introduced in Section 2.1. In other words, they convert external don't-cares into internal don't-cares before optimizations are performed. Although effective, this approach has the following drawbacks. First, the block boundaries are not preserved after optimization, which may make verification difficult, especially when dealing with third-party IP blocks in an SoC design. Second, dissolving boundaries makes it difficult to use external don't-cares because the chip's environment often depends on applications and cannot be modeled easily using a netlist. While state-of-the-art synthesis tools mostly exploit internal don't-cares, our work shows how to effectively exploit external don't-cares without viewing them as internal don't-cares and without blending multiple blocks into one netlist.

4. CIRCUIT CUSTOMIZATION FLOW AND NEW APPLICATIONS

In this section we describe flows that reuse existing simulation-based verification environments for circuit customization, including direct tests and constrained-random testbenches. Since direct tests provide all the test patterns in the care-set of the circuit, the techniques described in Section 3 can be applied directly. However, sometimes the inputs may only be partially known. For example, although the program running on an embedded system may be given, its input data may vary at runtime. To address this problem, we propose an innovative technique that uses the constrained-random testbench developed in most design verification flows as a "synthesis IP" for circuit customization. This approach guarantees that whatever verified by the testbench will still be correct in the customized circuit, even when some inputs are not given in advance.

This section is organized as follows. We first describe our circuit customization flow using constrained-random testbenches and propose a new verification method suitable for

this flow. We then finalize this section by proposing several applications enabled by our resynthesis techniques.

4.1 Circuit Customization Using Constrained-Random Testbench

Our circuit-customization flow using constrained-random testbenches works as follows. (1) Simulate the testbench for a certain number of cycles to produce a direct test. (2) Use the techniques described in the previous section to customize the circuit. (3) Verify the correctness of the circuit with respect to the testbench after each circuit modification in step (2) and only accept changes that passes verification. The verification step will be described in Section 4.2.

4.2 Verification of Circuit Customization Changes

Although many verification techniques can perform complete sequential equivalence checking between two circuits, such as reachability analysis and unbounded model checking [Ganai and Gupta 2007], they may not be scalable enough to handle today's designs. To address this problem, we describe a new algorithm to verify the correctness of a customized circuit with respect to a constrained-random testbench. The algorithm is based on symbolic simulation and bounded model checking, and it utilizes proof-by-induction to achieve complete proof. Due to its bounded nature, the algorithm can be applied to much larger designs than traditional techniques. The algorithm is shown in Figure 6. In the algorithm, *ckt1* is the original circuit, *ckt2* is the customized circuit, *tb* is the testbench and *n* is the number of cycles to be simulated. Function *verify* then checks if *ckt1* and *ckt2* produce identical results at *checker variables* within *n* cycles under the given constraints, whereas a checker variable is typically a primary output or a register in the circuit. Note that to achieve complete proof, we replace scalar random values in the testbench with symbols in line 4 to make sure all possible inputs are verified in our approach.

```
function verify(tb, ckt1, ckt2, n)
 1   initialize the circuit to a known symbolic state;
 2   repeat n cycles
 3     foreach random value v generated in tb
 4       replace v with a symbol;
 5     symbolically simulate one cycle;
 6     collect logic expressions generated at checker variables
       in ckt1 and ckt2;
 7   check equivalency of expressions of checker variables
     between ckt1 and ckt2;
 8   if all the expressions are equivalent
 9     return true;
10   else
11     return false;
```

Fig. 6. Circuit verification using symbolic simulation and constrained-random testbenches.

Ideally, initial symbolic state should be the set of all reachable states encoded symbolically. However, reachability analysis may be impossible for even moderate-size designs.

Therefore, in this work we assign pure scalar (over-constrained) or pure symbolic (under-constrained) values to the state bits depending on how those bits are used. If the verification algorithm returns false, then we abandon the change made to the circuit. Although we may lose some optimization opportunities because part of the state bits are under-constrained, this step is necessary to ensure the scalability of our verification method. If the verification algorithm returns true, then due to the over-constrained state bits, proof-by-induction should be used to generate additional rules for the constrained-random testbench to ensure the equivalency between $ckt1$ and $ckt2$ for *all* cycles, and the rules are derived as follows. Suppose that the initial state is called $state_i$ and the final state is called $state_f$. If $state_i \supseteq state_f$, then no further constraints are needed, and $ckt1$ and $ckt2$ will produce identical outputs for all the inputs that can be generated by the constrained-random testbench if the circuits are both initialized to $state_i$. On the other hand, if $state_i \subset state_f$, then additional constraints must be added to make sure $state_i$ is reached every n cycles. For example, if a pipelined processor is initialized to a state in which all general registers are symbols and all bypass control registers are 0. Further assume that algorithm $verify$ successfully confirmed the equivalency between $ckt1$ and $ckt2$ for 100 cycles. Then as long as the program running on the customized circuit makes all bypass control registers 0 every 100 cycle, both circuits will produce the same outputs. Note that if $state_i \cap state_f$ is empty, then proof by induction fails and the customized circuit is correct only within the simulated cycles.

From the analysis above, it can be observed that symbolically simulating more cycles will provide more flexibility for circuit customization. Typically, simulating cycles equal to twice the number of pipeline stages will yield good results because most inputs will be able to propagate through the pipeline. Note that if a wire remains 0 or 1 throughout symbolic simulation, then the wire can be replaced by the constant, and this can often initiate a chain of further optimizations.

4.3 New Applications

In this subsection we discuss some of the new applications that are enabled by our techniques, including three applications based on circuit specialization followed by one that requires synthesizing truth tables.

Acceleration of common-case computations: certain classes of SoC designs include several instances of a computational module to improve the parallelism of the system. For instance, this is the case for multimedia SoC where the required output throughput is achieved by increasing the parallelism of the computation. Among CPU designs, a specific example is the case of the Sun Niagara T1 where 8 processor cores were sharing one Floating Point Unit (FPU). However, due to its poor performance on FP testbenches, the second generation processor has been enhanced with 8 FPUs. Often the input distribution of components embedded in a system is highly skewed for a very small set, while remaining combinations are rare [Schnarr and Larus 1998]. For instance, it is observed that often under 10% of a program's instructions account for 90% of its execution time [Lakshminarayana et al. 2001]. Hence, SWEDE can be adopted to explore a "Better Than Worst-Case Design" methodology [Austin et al. 2005], also known as "Common-Case Computation" [Lakshminarayana et al. 2001], where one of several units is fully functional, and all others are optimized to only operate correctly for a few commonly-occurring input combinations. This approach reduces power and area of the final system. If an optimized computation fails at runtime, a fully-functional module is invoked as a back-up. Note that, for this ap-

proach to be viable, it may be necessary to deploy either a functional checker (validating the operation results) or a "valid input detection" circuit, as we are planning to explore. Alternatively, if one is only concerned with correctness on a small subset of inputs, faster circuits are possible as well. For example, the main idea in [Verma et al. 2008] is that much faster arithmetic circuits can be designed by allowing a small fraction of incorrect results. In contrast, we focus on circuit size rather than performance.

Customization of third-party IP components in an SoC: in order to improve reuse, SoC designs often acquire some components from third-party vendors. In the fourth quarter of 2007, total IP revenue has reached $265.4 million, with a growth rate of 4.1% each year [EETimes 2008]. Such components are typically embedded in an environment that only exploits a small fraction of their functionality. It is then possible to use SWEDE to reduce the component's complexity (and power consumption) based on the specific environment in which it is embedded. For example, floating point logic in an embedded processor is redundant if the target application does not require any floating point computation. Manually removing redundant portions of the design, however, can be difficult and error-prone. While some hard IPs are difficult to modify, a large segment of the $1B/year IP market consists of soft IPs, such as ARM processors, USB and PCI-Express devices, etc. The source code is given to customers unencrypted because design companies would not agree to put unknown blocks in their chips. In addition, design houses often need to patch possible problems and better optimize their entire SoC designs in terms of placement and floorplanning. Importantly, such source code can be modified, and the techniques in our paper may lead to new business models — competing on cost by simplifying existing IPs automatically. For example, there are many USB and PCI-Express peripherals for PCs and laptops that are dedicated to a single function, like WiFi, WiMax, voice-over-IP, Dolby 7.1 sound, etc. Needless to say, such devices do not exercise the entire bus protocol, but the IP on which they are built may support it. Therefore, to reduce the cost, one may automatically customize the inherited bus IP to a given application. Whether or not the cost differential is significant, IP specialization may noticeably reduce power consumption. For example, Apple iPhone contains the S-Gold2 baseband chipset from Infineon in which Apple chose to turn off FM radio support and MMC/SD card compatibility, apparently to reduce power [Walko 2007].

Graceful wear-out of electronic devices: extreme transistor scaling is leading to reduced silicon reliability, including early device and interconnect wear-out. To overcome the impact of this issue there is a growing need for low-cost reliable design solution. The use of SWEDE enables reliability through component sparing [Constantinides et al. 2006], where spare components can be optimized to provide only bare-bone functionality, sufficient to keep the system operational in critical aspects until it is replaced. An example of this spare-optimization application is discussed by Wagner et al. [2006], where the authors identify a small subset of a processor design that must be kept operational in order to provide full system functionality (in this case the spare was part of the processor itself with acceleration features excluded). When the original circuit becomes unreliable, it will be replaced by the barebone spare component to avoid a system-level crash.

Synthesis for fast emulation: in the emulation domain, one common issue is the synthesis of the input constraints. Emulation systems can apply constrained-random simulation at very high performance compared to logic simulation. However, if the input constraints are not synthesizable, then at each clock cycle the emulator must communicate with a

simulating host, incurring a huge performance impact on the emulation. At the same time, input constraints are often written in a high-level language (C++, Vera, etc.) and cannot be synthesized. SWEDE can be deployed by running the random simulation *only* on the design's input constraints (and not including the design itself). This simulation would be very fast and generates a set of care terms that SWEDE then synthesizes in a circuit uploaded on the emulator along with the design. Each emulation run would use a different constraint circuit, each synthesized by SWEDE based on the random stimuli. On the other hand, the design itself does not need to be resynthesized for each run.

5. EXPERIMENTAL RESULTS

In this section, we use three design examples to evaluate the capability of SWEDE in customizing circuits: an Alpha processor running real applications, an integer multiplier, and a DLX processor with constrained-random testbenches. In addition, we compare SWEDE with existing synthesis tools to evaluate its ability to synthesize truth tables with external don't-cares. These tools are Espresso, MVSIS and a commercial synthesis tool. Table I reports the numbers of primary inputs and outputs, as well as initial cell count for the benchmarks used. Benchmarks C1908-C7552 are from ISCAS'85. Both Alpha and DLX are processors from the Bug UnderGround project [Bertacco et al. 2007] that implement subsets of the Alpha and MIPS ISA, respectively. Our experiments were performed on Linux workstations with AMD Opteron 280 CPUs (2.4GHz) equipped with 8GB of memory.

Table I. Characteristics of benchmarks.

Benchmark	Description	#In/Outputs	#Cells
C1908	16-bit SEC/DED circuit	33/25	461
C2670	12-bit ALU and controller	233/140	484
C3540	8-bit ALU	50/22	1060
C5315	9-bit ALU	178/123	1057
C7552	32-bit adder/comparator	207/108	1187
Alpha	5-stage pipeline Alpha CPU	3054/3619	30531
DLX	5-stage pipeline MIPS-lite CPU	2127/2160	14725
Multiplier	16-bit Wallace tree multiplier	32/32	1938

5.1 Case Studies

Case study 1 (Alpha processor): for this study we ran five applications from the SpecINT [2000] suite, whose characteristics are summarized in Table II. The processor was synthesized using Cadence RTL compiler with the highest optimization effort, and was mapped to a 0.18μm library. Since our Alpha processor only implements a subset of the Alpha ISA, simulation was performed in lockstep with the Simplescalar instruction set simulator [Austin et al. 2002]. We then use SignalMerge to optimize the circuit based on the stimuli from each program. Figure 7 and 8 report the final sizes of the optimized designs and the synthesis runtimes, respectively, achieved after simulating up to half a million instructions. They indicate that the optimization potential varies from application to application: for instance, the bzip2 application has a very small stimuli set, hence we can exploit aggressive

optimizations on it; while gcc has a much wider span, hence little optimization can be extracted. This is aligned with the intuition that bzip2 is a specialized algorithm applying the same operations to arbitrary data sets, while gcc's operation is much more complex. This result suggests that if the program running on a circuit is known, SWEDE can potentially reduce its size significantly, generating a much smaller circuit that consumes less power. Figure 8 also shows that SignalMerge operates in approximately linear time on the number of input vectors in the care set, which enables it to handle complex designs efficiently. Designs can be further optimized by ShannonSynth: this step has greater runtime complexity; however, this is offset by the fact that ShannonSynth only takes into consideration small blocks in a circuit. For comparison, in the figures we also show the trend of optimizing for a constrained-random trace generated by StressTest [Wagner et al. 2005] (diamond-bullet lines). Its curve indicates that with random inputs, we can only reduce the circuit by 10%, even when the number of instructions is as small as 6400. This is not surprising since, intuitively, random traces span a much larger fraction of the circuit's configurations than real applications, making optimization difficult.

Table II. Characteristics of SpecINT programs [SpecINT 2000].

Benchmark	Description	Language
bzip2	Compression tool	C
gcc	Compiler	C
mcf	Combinatorial optimization	C
parser	Word processing	C
perlbmk	Perl programming language	C

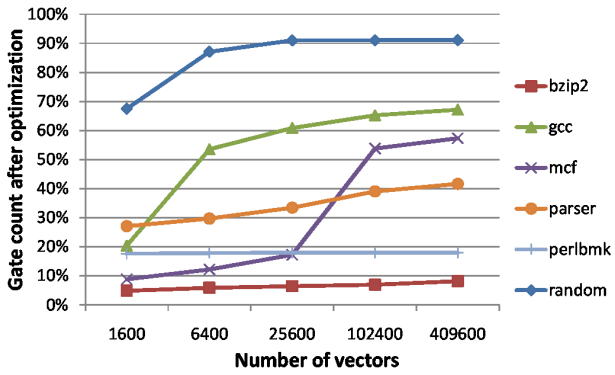

Fig. 7. Gate count after specializing the Alpha CPU with SignalMerge. 30-90% of the gates can be removed for applications as long as half-million dynamic instructions.

In Figure 9 we show the results when optimizing individual components in the Alpha processor using the gcc application. The blocks we studied are the instruction fetch unit (IF), the decode unit (ID), the execute block (EX) and the memory access controller (MEM). The result indicates that the optimization potential is very block specific. In particular, the EX block cannot be optimized well because the execution unit needs to handle

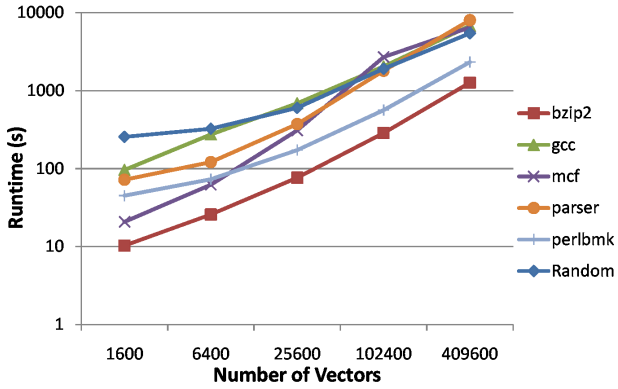

Fig. 8. SignalMerge runtime to specialize Alpha. Runtime is approximately linear on the number of stimulus vectors used.

a wide range of input values, making don't-cares less dense. The MEM block also has very limited optimization potential because it only has 363 gates but has 195 inputs. This shallow logic structure makes signal sharing difficult.

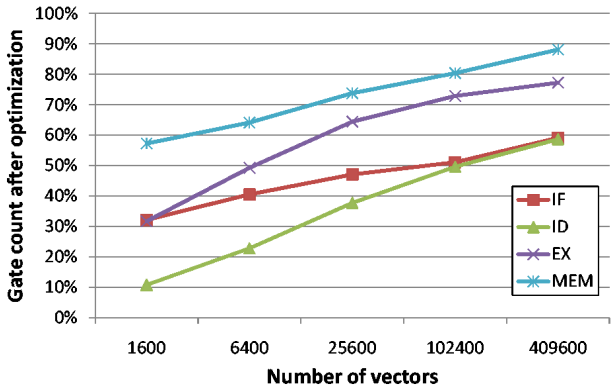

Fig. 9. Gate count of Alpha blocks after specialization.

Case study 2 (constant-coefficient multiplier): embedded systems and digital signal processors often need to perform simple operations repetitively [Lai et al. 2008; Sarbishei et al. 2009]. For example, consider a portable electronic measurement device that must convert between US units and metric units while keeping power consumption low. To keep the circuit simple, an integer multiplier can be used, adjusting the decimal point afterward.

To support conversions between inches, feet, miles and meters, one needs to be able to multiply by the following six constants: 2.54, 30.4, 1.61 and their inverse. For the sake of this example, we made the assumption that the user can only compute with 5-digit decimal values. We used SWEDE to optimize the circuit starting from a 16-bit Wallace-tree multiplier. The original circuit had 1938 gates, and our care set included 393216 patterns. For comparison, we converted external DCs into internal DCs by hard-coding the constants in

the RTL code, and then we synthesized the design using two different commercial synthesis tools, Tool1 and Tool2. The results are summarized in Table III. Since different synthesis tools may use different multiplier architectures, the reduction ratios should be compared instead of the cell counts. As the results suggest, FastShrink performs better than existing synthesis tools. For comparison with existing tools that support true external don't-care synthesis, we also attempted to synthesize the truth table of the 393216 patterns using Espresso and Tool1 (truth-table synthesis mode) but could not obtain a result netlist after 96 hours.

Table III. Comparison of two major commercial tools and SWEDE in synthesizing constant-coefficient multipliers. Original cell counts, optimized cell counts and the reduction ratios are shown. FastShrink runtime was 42 seconds.

	Tool1		Tool2		**FastShrink**	
	Orig.	Opt.	Orig.	Opt.	Orig.	Opt.
Cell count	1387	834	2238	1440	1938	981
Reduction Ratio	39.9%		35.7%		**49.4%**	

While this multiplier only serves as a simple and intuitive example, the case study indicates that SWEDE can seamlessly handle even traditionally difficult synthesis problems, such as multipliers. This is because SWEDE is unconcerned with the complexity of the original functionality and can focus on just a few important inputs for its optimization. This characteristic makes SWEDE considerably different from domain-specific optimization techniques such as [Sarbishei et al. 2009] in that our methods do not require architectural information. To further study the behavior of the specialized multiplier, we computed all the multiplications where one input ranges from 0 to 65535, and the other from 100 to 199, producing a total of 6553600 input combinations. The range for the second input was selected around the range of our specialized input constants. The results show that 29.33% of the input combinations were still multiplied correctly, while the average error over all input combinations was 9.75%. The greatest error we observed was 98.72%, produced by 56685×188.

Case study 3 (customizing DLX with a constrained-random testbench): in this case study we customize DLX with constrained-random testbenches that allow the use of different combinations of instructions. Insight [Avery 2008], a commercial symbolic simulator that can symbolically simulate behavior-level testbenches as well as gate-level netlists, is used in this case study. The circuit is initialized to a state in which all general registers are symbols and all control registers are scalar values. We then prepare four testbenches that generate different combinations of instructions with random data values, and the number of cycles used in verification is 10. In this case study, we report the numbers of registers that are proven to be constant under different testbenches. Those registers can then be removed to simplify the circuit, and the results are summarized in Table IV. The results suggest that when fewer numbers of instructions are used, more logic becomes redundant and can be removed. Since we assign random values to data inputs in the testbenches, the customized circuit will produce correct outputs for any input as long as the instructions used in the program comply with those used in the testbenches and the control registers return to their initial scalar values every 10 cycles. The latter condition can be achieved by inserting a few NOPs before the 10-cycle boundary. This case study also suggests that by

developing different constrained-random testbenches to model different usages, SWEDE can generate various customized circuits to measure the trade-off among the functionality of a circuit, the die area and its power consumption. Note that while previous case studies only focus on optimizing the combinational part of the circuits, in this case study we actually performed a simple form of sequential optimization because some registers are removed.

Table IV. Percentage of registers that can be removed using different combinations of instructions and random data inputs. Runtime is the time for checking whether a register is constant.

Instructions allowed	Register reduction	Runtime (sec)
NOP	60.4%	1
ADD, ADDI, NOP	33.9%	8
ADD, ADDI, LW, SW	31.9%	12
ADD, ADDI, LW, SW, SLL, SRA, BEQ, ORI	10.1%	37

5.2 Comparison with Existing Tools

In this experiment we compared CleanSlate and InterSynth with Espresso and a commercial tool (Tool1). We used the ABC system [ABC 2007] to implement the interpolation-based procedure InterSynth for computing multi-level representations of Boolean functions that agree with the given on-set/off-set. The results are verified by checking that interpolants are implied by the on-sets and do not overlap with the off-sets. To avoid the influence of technology mapping on our experiments, we only used inverters and basic two-input gates. To evaluate Espresso, which lacks a technology mapper, we fed the optimized truth tables to ABC. We used 128 random patterns to generate the truth tables, and summarized the results in Table V. CleanSlate and InterSynth outperform Espresso and Tool1, producing the smallest netlists in just a small fraction of the time. Moreover, in several cases Tool1 timed-out after one hour. We also tried synthesizing from care sets of 256, 512 and 1024 random patterns using the same circuits. We found that CleanSlate can finish all the benchmarks within 6.5 minutes, while Espresso and Tool1 timed-out after 1 hour for most of the benchmarks.

To compare CleanSlate and InterSynth with traditional techniques based on decision diagrams and sum-of-product manipulations, we conducted another experiment that optimizes the truth tables using the MVSIS *mfs* command [MVSIS 2005]. Since MVSIS requires don't-care terms to be explicitly specified in the input file when the PLA format is used, we reduced the truth tables to include only the first 16 inputs so that the file sizes were reasonable. The results are summarized in Table VI. From the table we can observe that MVSIS often produces the smallest netlists. However, since runtime is also significantly longer, this solution cannot scale to large designs. The results also indicate that CleanSlate outperforms InterSynth in every instance, suggesting that CleanSlate may be more suitable for optimizing truth tables with fewer inputs.

Note that the complexity of the interpolation procedure, in the worst case, is the same as that of Boolean satisfiability for circuit-based problems: exponential in the number of input variables of the circuit and in the number of logic levels. However, in most of the

Table V. Comparison of existing tools and SWEDE using one-hour time-out. All our solutions, CleanSlate, InterSynth and FastShrink, provide better synthesis quality with significantly shorter runtime.

Benchmark	Number of cells after (re)synthesis				
	Truth table based				Netlist based
	Espresso	Tool1	CleanSlate	InterSynth	FastShrink (SignalMerge)
C1908	2518	6891	**1352**	**828**	**284** (332)
C2670	6098	T/O	**4467**	**2592**	**571** (665)
C3540	1925	6271	**1140**	**1980**	**1059** (1094)
C5315	5183	T/O	**3594**	**5882**	**1238** (1312)
C7552	5072	T/O	**3644**	**4923**	**1311** (1387)

Benchmark	Runtime (s)				
	Truth table based				Netlist based
	Espresso	Tool1	CleanSlate	InterSynth	FastShrink (SignalMerge)
C1908	16.19	143.76	**4.17**	**0.99**	**33.68** (0.32)
C2670	1494.51	T/O	**45.26**	**34.81**	**54.13** (1.36)
C3540	29.12	193.69	**3.55**	**2.01**	**115.4** (1.54)
C5315	635.17	T/O	**27.70**	**25.04**	**179.56** (1.42)
C7552	911.54	T/O	**35.39**	**26.68**	**150.51** (0.71)

Table VI. Comparison of MVSIS and SWEDE. Due to input format limitations of MVSIS, the truth tables were reduced to contain only 16 inputs.

Benchmark	Number of cells after synthesis			Runtime (s)		
	MVSIS	CleanSlate	InterSynth	MVSIS	CleanSlate	InterSynth
C1908	773	**1485**	**1693**	37.20	**0.14**	**1.79**
C2670	4053	**4675**	**9188**	210.36	**0.37**	**6.30**
C3540	814	**1232**	**1652**	32.33	**0.12**	**1.76**
C5315	3425	**3757**	**7508**	203.79	**0.41**	**4.54**
C7552	3443	**3820**	**7355**	166.01	**0.41**	**4.34**

practical cases, it works well because the number of conflicts (the metric that determines the number of resolution steps and, therefore, the initial size of the interpolant) is relatively small. For the designs synthesized by InterSynth in this experiment, there were no more than 5,000 conflicts, which led to initial interpolants whose size did not exceed 50,000 AIG nodes.

Although CleanSlate and InterSynth, which operate from a truth table specification, produce comparatively better results than Espresso and commercial tools, a comparison between Table V and Table I shows that the generated netlists are still larger than the original ones. The reason is that the original netlists are often produced from higher-level specifications, which include conceptual structures that lead to better optimizations. On the other hand, trying to synthesize a compact netlist using only input and output values is much more difficult. Therefore, if a netlist is available, the best optimizations can be obtained through FastShrink, whose results are also shown in Table V.

SWEDE is based on signatures, which can be calculated easily using simulation. This

makes SWEDE simple to use because designers only need to provide input vectors to the circuit that belong to the care terms. Since signatures can be represented compactly using bit-vectors and allow bit-parallel computation, our solution is both fast and memory-efficient. As our experimental results show, we can handle half-million input vectors in less than three hours.

6. CONCLUSIONS

To reduce circuit design complexity in the multi-billion transistor era, SoC and embedded systems heavily rely on reuse and third-party IP components. Often, the design environment surrounding such components uses only a fraction of the functionality that these general-purpose components implement. The unused logic in those circuit blocks not only occupies valuable die area but also consumes more power, hurting the circuit's performance and quality. Hence, new synthesis optimization opportunities are available in simplifying these components to the subset of functionality required by the system they are embedded in. Surprisingly, existing synthesis tools perform poorly in this context, which typically involves a small care-set and a very large don't-care set. To address this problem, we proposed a new tool called SWEDE, and provided three new synthesis techniques which can specialize a circuit using external don't-cares: FastShrink, CleanSlate and InterSynth. Unlike traditional synthesis tools that pursue maximal use of don't-cares by explicitly branching on different don't-care assignments, our greedy algorithms (SignalMerge and the first phase of CleanSlate) implicitly exploit the fact that most terms are don't-cares and quickly generate a small netlist. Further circuit optimization is performed by our ShannonSynth technique and the second phase of CleanSlate. This novel synthesis flow allows SWEDE to scale better when massive don't-cares exist. In addition, SWEDE can reuse existing verification environments, such as direct test or constrained-random testbenches, for circuit customization. Therefore, it can make sure whatever verified by the testbenches is still correct in the customized circuits. Since such testbenches exist in most verification flows, SWEDE can be adopted easily in most designs. As our empirical results indicate, SWEDE provides comparable or better synthesis quality than state-of-the-art tools while running 10X faster. In fact, SWEDE can handle designs as large as 30K cells with 0.5M care-set vectors in a few hours, demonstrating its superior scalability and efficiency.

We discussed a number of new applications enabled by SWEDE, including new system-design paradigms and solutions to current engineering problems. These new applications promise to produce circuits that run faster, consume less power, and can be used as inexpensive back-up modules for larger circuits that may fail during operation.

REFERENCES

ABC. 2007. Berkeley logic synthesis and verification group, ABC: A system for sequential synthesis and verification, release 80308.

AUSTIN, T. M., BERTACCO, V., BLAAUW, D., AND MUDGE, T. N. 2005. Opportunities and challenges for better than worst-case design. In *ASP-DAC*. 2–7.

AUSTIN, T. M., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2, 59–67.

AVERY. 2008. Avery Design Systems, company web site http://www.avery-design.com/.

BERTACCO, V. 2005. *Scalable Hardware Verification with Symbolic Simulation.* Springer.

BERTACCO, V., AUSTIN, T., AND WAGNER, I. 2007. Bug UnderGround project, http://bug.eecs.umich.edu/.

BRAYTON, R. K., GAO, M., JIANG, J.-H. R., JIANG, Y., LI, Y., MISHCHENKO, A., SINHA, S., AND VILLA, T. 2002. Optimization of multi-value multi-level networks. In *ISMVL*. 168–177.

CHANG, K.-H., BERTACCO, V., AND MARKOV, I. L. 2009. Customizing IP cores for system-on-chip designs using extensive external don't-cares. In *DATE*. 582–585.

CHOUDHURY, M. R. AND MOHANRAM, K. 2009. Masking timing errors on speed-paths in logic circuits. In *DATE*. 87–92.

CONSTANTINIDES, K., PLAZA, S., BLOME, J. A., ZHANG, B., BERTACCO, V., MAHLKE, S. A., AUSTIN, T. M., AND ORSHANSKY, M. 2006. BulletProof: a defect-tolerant CMP switch architecture. In *HPCA*. 5–16.

CRAIG, W. 1957. Linear reasoning: a new form of the Herbrand-Gentzen theorem. *Jour. of Sym. Logic 22*, 3, 250–287.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *SAT*. 502–518.

EETIMES. Apr. 03, 2008. EDA sales jump in Q4, http://www.eetimes.com/showarticle.jhtml?articleid=207001548. In *EETimes*.

GANAI, M. K. AND GUPTA, A. 2007. *SAT-based Scalable Formal Verification Solutions*. Springer.

GORJIARA, B. AND GAJSKI, D. 2008. Automatic architecture refinement techniques for customizing processing elements. In *DAC*. 379–384.

LAI, C.-Y., HUANG, C.-Y., AND KHOO, K.-Y. 2008. Improving constant-coefficient multiplier verification by partial product identification. In *DATE*. 813–818.

LAKSHMINARAYANA, G., RAGHUNATHAN, A., KHOURI, K. S., AND JHA, N. K. 2001. Method for synthesis of common-case optimized circuits to improve performance and power dissipation. *United States Patent* 6,308,313 B1.

MATSUNAGA, Y. AND FUJITA, M. 1989. Multi-level logic optimization using binary decision diagrams. In *ICCAD*. 556–559.

MCMILLAN, K. L. 2003. Interpolation and SAT-based model checking. In *CAV*. 1–13.

MISHCHENKO, A., BRAYTON, R., JIANG, J.-H. R., AND JANG, S. 2007. SAT-based logic optimization and resynthesis. In *IWLS*. 358–364.

MUROGA, S., KAMBAYASHI, Y., LAI, H. C., AND CULLINEY, J. N. 1989. The transduction method-design of logic networks based on permissible functions. *IEEE Trans. Computers 38*, 10, 1404–1424.

MVSIS. 2005. http://www-cad.eecs.berkeley.edu/respep/research/mvsis.

PLAZA, S., CHANG, K.-H., MARKOV, I. L., AND BERTACCO, V. 2007. Node mergers in the presence of don't cares. In *ASP-DAC*. 414–419.

RAJSKI, J. AND VASUDEVAMURTHY, J. 1992. The testability-preserving concurrent decomposition and factorization of Boolean expressions. *IEEE Trans. on CAD of Integrated Circuits and Systems 11*, 6, 778–793.

RUDELL, R. L. AND SANGIOVANNI-VINCENTELLI, A. L. 1987. Multiple-valued minimization for PLA optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems 6*, 5, 727–750.

SARBISHEI, O., TABANDEH, M., ALIZADEH, B., AND FUJITA, M. 2009. High-level optimization of integer multipliers over a finite bit-width with verification capabilities. In *MEMOCODE*. 56–65.

SAVOJ, H. AND BRAYTON, R. K. 1990. The use of observability and external don't cares for the simplification of multi-level networks. In *DAC*. 297–301.

SCHNARR, E. AND LARUS, J. R. 1998. Fast out-of-order processor simulation using memoization. In *ASPLOS*. 283–294.

SHANNON, C. E. 1948. A mathematical theory of communication. *The Bell System Technical Journal 27*, 379–423.

SPECINT. 2000. SpecINT2000 benchmarks, http://www.spec.org/.

VERMA, A. K., BRISK, P., AND IENNE, P. 2008. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *DATE*. 1250–1255.

WAGNER, I., BERTACCO, V., AND AUSTIN, T. M. 2005. StressTest: an automatic approach to test generation via activity monitors. In *DAC*. 783–788.

WAGNER, I., BERTACCO, V., AND AUSTIN, T. M. 2006. Shielding against design flaws with field repairable control logic. In *DAC*. 344–347.

WALKO, J. Jul. 02, 2007. Europe suppliers score in Apple's iPhone, http://eetimes.eu/showarticle.jhtml?articleid=200001829. In *EETimes Europe*.

YAMASHITA, S., SAWADA, H., AND NAGOYA, A. 1996. A new method to express functional permissibilities for LUT based FPGAs and its applications. In *ICCAD*. 254–261.

YAMASHITA, S., SAWADA, H., AND NAGOYA, A. 2000. SPFD: A new method to express functional flexibility. *IEEE Trans. on CAD of Integrated Circuits and Systems 19*, 8, 840–849.

YANG, Y.-S., SINHA, S., VENERIS, A. G., AND BRAYTON, R. K. 2007. Automating logic rectification by approximate SPFDs. In *ASP-DAC*. 402–407.