Customizing IP Cores for System-on-Chip Designs Using Extensive External Don't-Cares

Kai-hui Chang*‡, Valeria Bertacco* and Igor L. Markov*†
*EECS Department, University of Michigan, Ann Arbor, MI
‡Avery Design Systems, Andover, MA
†Synopsys, Inc., Sunnyvale, CA
{changkh, valeria, imarkov}@umich.edu

***Abstract*—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 reduce its area and power consumption. Therefore we extend modern digital synthesis with a novel technique, called SWEDE, that uses external 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.**

I. Introduction

Due to the increasing demand for integrated circuits to provide more functions while consuming less power, designing a new chip becomes more difficult. To reduce this design effort, it is common 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. Since the reused components often have specific target applications and environment, system performance and cost may be improved by removing logic that is not invoked in such environments. However, this novel optimization, which we call *design customization*, poses a new synthesis challenge which is different from traditional formulations by the abundance of external don't-cares. Our experimental study revealed that the performance of existing tools greatly deteriorates when extensive don't-cares are added. In addition, several tools do not provide specification formats for this situation. The latter is especially problematic because without an efficient way to represent such don't-cares for synthesis tools, the adoption of circuit-customization methodologies will be hindered.

To utilize abundant external don't-cares that exist when an IP is embedded in a System-on-Chip (SoC) design for circuit customization, we developed a FastShrink algorithm. This algorithm takes an existing design as input and reduces it based on the specified don't-care set. Our second contribution is a framework 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 into a tool called *SWEDE* (Synthesis Within an Extensive Don't-care Environment). Since SWEDE reuses testbenches developed for verification to customize circuits, it can make sure whatever verified by the testbenches is still correct after customization. We empirically compared SWEDE with existing synthesis tools and observed SWEDE producing smaller circuits.

SWEDE's high performance enables several new synthesis applications and enhances many others, including (1) customization of third-party IP components in an SoC; (2) acceleration of the most-frequent computation in a unit [1], [5]; and (3) support for graceful wear-out of electronic devices [8]. These applications 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 [8], and lower-power circuits [5]. 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.

The rest of the paper is organized as follows. In Section II we review previous work and provide necessary background. Section III describes our new techniques, whose verification methods are given in Section IV. Experimental results are provided in Section V, and Section VI concludes this paper.

II. Background and Previous Work

A. Previous Work

Several existing techniques perform circuit customization using don't-cares via resynthesis such as rewiring [10], [11] and node merging [6]. Recently, Gorjiara *et al.* [3] 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.

B. Bit-Signatures and Entropy

Our FastShrink technique is based on bit-signatures generated using simulation, where a bit in a signature is the simulation value of an input vector. The second step of the FastShrink technique (see Section III-B) 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* defined as follows [7]:

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

In (1), 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. A larger E means that the signature contains more information.

C. Simulation and Proof by Induction

Simulation is the most popular verification method: 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, while in *symbolic simulation* symbols are used. 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. One way to solve this problem is to use proof by induction [2]. 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.

III. Circuit Customization with Direct Test

In this section we formalize the synthesis problem described earlier and propose two circuit-optimization techniques.

A. Problem Formulation

Given a circuit and the complete set of all possible input vectors (i.e., a direct test), we seek to produce a small netlist that generates the correct outputs for the given inputs.

B. Customizing an Existing Netlist

Given an existing netlist, FastShrink uses a two-step process to produce a customized 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 1. It first simulates care-term vectors from a direct test and then merges signals with identical signatures. This allows SignalMerge to leverage both external and internal satisfiability don't-cares to remove redundant gates. 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. 1. The SignalMerge 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. 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.

```
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 Espresso;
7     if (resynthesized netlist is smaller)
8       replace the subcircuit with the resynthesized netlist;
```

Fig. 2. The ShannonSynth algorithm.

The ShannonSynth algorithm in Figure 2 first simulates vectors in the care terms to generate a signature for each signal. Next, it computes the entropy of each signature and 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 Espresso and then performs further optimization using ABC [12]. If the new resynthesized netlist is smaller than the original one, ShannonSynth replaces it.

C. 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. 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.

IV. Circuit Customization with Constrained-Random Testbench

The techniques described in Section III performs well when the inputs to a circuit are completely known. However, sometimes the inputs may only be partially known, such as input data to a program. To address this problem, we propose an innovative technique that uses the constrained-random testbench developed in most design verification flows 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.

A. Circuit Customization Flow

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 IV-B.

B. Verification of Customized Circuit

Although many verification techniques can perform complete sequential equivalence checking between two circuits, such as reachability analysis and unbounded model checking [2], 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 3. 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   return (all the expressions are equivalent) ? true : false;
```

Fig. 3. Circuit verification using constrained-random testbenches.

If the verification algorithm returns false, then we abandon the change made to the circuit. If the verification algorithm returns true, 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.

V. Experimental Results

In this section, we use three design examples to evaluate the capability of SWEDE: an Alpha processor running real applications, an integer multiplier, and a DLX processor with a constrained-random testbench. Table I reports the characteristics of the benchmarks. Alpha and DLX are from [14] that implement subsets of the Alpha and MIPS ISA, respectively. Our experiments were performed on Linux workstations with AMD Opteron 280 CPUs (2.4GHz) and 8G memory.

TABLE I
Characteristics of benchmarks.

Benchmark	Description	#Cells
Alpha	5-stage pipeline Alpha CPU	30531
DLX	5-stage pipeline MIPS-lite CPU	14725
Multiplier	16-bit Wallace tree multiplier	1938

Case study 1 (Alpha processor): for this study we ran five applications from the SpecINT'00 suite [15]. Benchmark bzip2 is a compression tool, gcc is a C compiler, mcf performs combinatorial optimization, parser does word processing, and perlbmk is Perl programming language. The processor was synthesized using Cadence RTL compiler. We then use SignalMerge to optimize the circuit based on the stimuli from each program. Figure 4 and 5 report the final sizes of the optimized designs and the synthesis runtimes, 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. Figure 5 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 [9] (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.

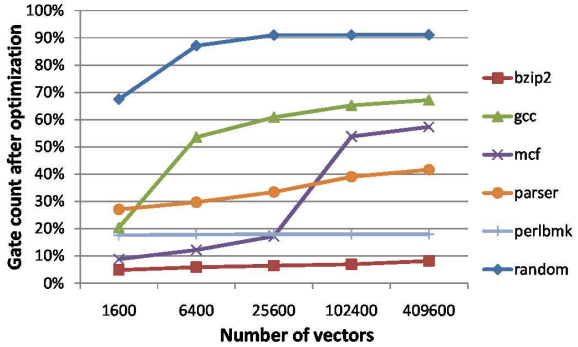

Fig. 4. Gate count after customizing the Alpha CPU with SignalMerge.

Case study 2 (constant-coefficient multiplier): embedded systems and digital signal processors often need to perform

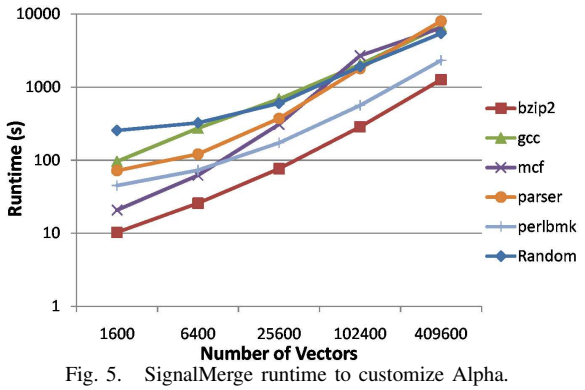

Fig. 5. SignalMerge runtime to customize Alpha.

simple operations repetitively [4]. 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 is used, adjusting the decimal point afterward. To support conversions between inches, feet, miles and meters, the following six constant multipliers are needed: 2.54, 30.4, 1.61 and their inverse. We further assume 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 393,216 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. The results are summarized in Table II. Since different 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 tools.

TABLE II
COMPARISON OF COMMERCIAL TOOLS AND SWEDE IN SYNTHESIZING CONSTANT-COEFFICIENT MULTIPLIERS.

	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%**	

To further study the behavior of the customized multiplier, we computed all the multiplications where one input ranges from 0 to 65535, and the other from 100 to 199. Among the input numbers, 29.33% were still multiplied correctly, while the average error 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 [13], a commercial symbolic simulator, was used in this case study. The circuit was initialized to a state in which all general registers were symbols and all control registers were scalar values. We then prepared four testbenches that generate different combinations of instructions with random data values, and the number of cycles used in verification was 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 III. The results suggest that when fewer numbers of instructions are used, more logic becomes redundant and can be removed. Since we assign symbolic 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 values every 10 cycles.

TABLE III
PERCENTAGE OF REGISTERS THAT CAN BE REMOVED USING DIFFERENT COMBINATIONS OF INSTRUCTIONS AND RANDOM DATA INPUTS.

Instructions allowed	Register reduction	Run time
NOP	60.4%	1s
ADD, ADDI, NOP	33.9%	8s
ADD, ADDI, LW, SW	31.9%	12s
ADD, ADDI, LW, SW, SLL, SRA, BEQ, ORI	10.1%	37s

VI. CONCLUSIONS

In this paper we proposed a new tool called SWEDE, and provided new synthesis techniques which can customize a circuit using external don't-cares. Unlike traditional synthesis tools that pursue maximal use of don't-cares by explicitly branching on different don't-care assignments, our SignalMerge algorithm implicitly exploits the fact that most terms are don't-cares and quickly generate a small netlist. Further circuit optimization is performed by our Shannon-Synth technique. This novel synthesis flow allows SWEDE to scale better when massive don't-cares exist. In addition, SWEDE reuses existing verification environments, such as direct test or constrained-random testbenches, for circuit customization. Since such testbenches exist in most verification flows, SWEDE can be adopted easily in most designs.

REFERENCES

[1] T. Austin, V. Bertacco, D. Blaauw and T. Mudge, "Opportunities and Challenges for Better Than Worst-Case Design", ASPDAC'05, pp. 2-7.
[2] M. K. Ganai and A. Gupta, "SAT-based Scalable Formal Verification Solutions", Springer, 2007.
[3] B. Gorjiara and D. Gajski, "Automatic architecture refinement techniques for customizing processing elements", DAC'08, pp. 379-384.
[4] C.-Y. Lai, C.-Y. Huang and K.-Y. Khoo, "Improving Constant-Coefficient Multiplier Verification by Partial Product Identification", DATE'08, pp. 813-818.
[5] G. Lakshminarayana, A. Raghunathan, K. S. Khouri and N. K. Jha, "Method for Synthesis of Common-Case Optimized Circuits to Improve Performance and Power Dissipation", United States Patent, No. 6,308,313 B1, Oct. 2001.
[6] S. M. Plaza, K.-H. Chang, I. L. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares", ASPDAC'07, pp. 414-419.
[7] C. E. Shannon, "A Mathematical Theory of Communication", The Bell System Technical Journal, Vol. 27, Oct. 1948, pp. 379-423.
[8] I. Wagner, V. Bertacco and T. Austin, "Shielding Against Design Flaws with Field Repairable Control Logic", DAC'06, pp. 344-347.
[9] I. Wagner, V. Bertacco and T. Austin, "StressTest: An Automatic Approach to Test Generation via Activity Monitors", DAC'05, pp. 783-788.
[10] S. Yamashita, H. Sawada, and A. Nagoya, "A New Method to Express Functional 1996, pp. 254-261.
[11] S. Yamashita, H. Sawada and A. Nagoya, "SPFD: A New Method to Express Functional Flexibility", IEEE TCAD, Aug. 2000, pp. 840-849.
[12] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 80308. http://www-cad.eecs.berkeley.edu/~alanmi/abc/
[13] http://www.avery-design.com/
[14] Bug UnderGround, http://bug.eecs.umich.edu/
[15] SpecINT2000 benchmarks, http://www.spec.org/