

# Scalably-Verifiable Sequential Synthesis

Robert Brayton

Alan Mishchenko

Department of EECS, University of California, Berkeley

{brayton, alanmi}@eecs.berkeley.edu

## Abstract

*This report is a case-study of the synergy between sequential synthesis and verification. Described is an efficient implementation of a sequential synthesis that uses simple induction to detect and merge sequentially-equivalent registers and nodes in a sequential circuit with a given initial state. Since retiming is not performed, state-encoding, scan chains, and test vectors are essentially preserved with minor modifications. Moreover, the synthesis results are guaranteed to be verifiable against the original circuits. Verification uses an inductive prover comparable to that used for synthesis, with the runtime close to that of synthesis. Experiments show that applying the synthesis flow to the largest academic benchmarks reduces both registers and area by more than 30%. Applying the flow to a set of 50 industrial benchmarks ranging in size from 100 to 20K registers shows an average reduction of 32% in registers and 15% in area while preserving the delay. The geometric averages of synthesis and verification runtime on the industrial benchmark set are close to 50 sec and 25 sec, respectively. The implementation is publicly available in the logic synthesis and verification system ABC.*

## 1 Introduction

Given a circuit with registers initialized to a given state, the set of reachable states are all possible valuations of the registers after arbitrarily long simulation from the initial state, by applying any values at the primary inputs. Two circuits are sequentially-equivalent if, in all reachable states, they produce equal output values in response to any values of the primary inputs.

Combinational synthesis involves changing the combinational logic of the circuit with no knowledge of its reachable states. As a result, the Boolean functions of the primary outputs and register inputs are preserved for any state of the registers. Combinational methods are useful because they offer some flexibility in modifying the circuit structure and can be easily verified. On the other hand, they have limited optimization power because the reachable state information is not exploited.

On the other hand, sequential synthesis modifies the circuit so that its behavior is preserved on the reachable states but arbitrary changes are allowed on the unreachable states. This is why, after sequential synthesis, the primary outputs and register inputs can differ as combinational functions expressed in terms of the register outputs and primary inputs. However, these functions are guaranteed to be unchanged for all reachable states, implying that the resulting circuit is sequentially-equivalent to the original one. Since for most practical circuits the set of reachable states is only a small fraction of all states, sequential synthesis allows for better logic restructuring, compared to the combinational synthesis.

Thus, when the combinational methods reach their limits, sequential synthesis becomes the next thing to try. This is

happening now because the design teams that traditionally used only combinational synthesis are turning to sequential synthesis for additional delay minimization and power reduction.

The traditional sequential synthesis (as exemplified by SIS) is predominantly structural. It performs register sweep, which merges stuck-at-constant registers, and register retiming, which moves registers over the combinational nodes while preserving the sequential behavior of the primary outputs.

Beside its limited nature, a significant drawback of this kind of synthesis is that it changes state encoding and therefore invalidates the initialization sequences and functional test-vectors developed for the original design. It was also shown that if retiming is performed in the middle of logic restructuring, proving sequential equivalence is, in general, PSPACE-complete [11].

Thus, traditional sequential synthesis based on register sweeping and retiming,

- has limited optimization power,
- invalidates state encoding, initialization sequences, and test-benches, and
- may be hard to verify.

This report is a case-study of sequential synthesis based on identifying pairs of sequentially-equivalent nodes (that is, nodes having the same or opposite values in all reachable states). These nodes can be merged without changing the sequential behavior of the circuit, leading to a substantial reduction of the circuit, as some pieces of logic are discarded because they no longer feed into the primary outputs. A well-developed methodology for K-step induction [9][4][24][18] can be used to efficiently compute pairs of sequentially-equivalent nodes.

We show that the proposed form of synthesis mitigates most of the above listed drawbacks of the traditional synthesis. Although there is no containment relation in expressive power between this synthesis and the traditional one, the proposed synthesis is quite powerful, as can be seen from the experimental results.

Unlike retiming, the proposed synthesis requires only minor systematic changes to the state encoding, init-sequences, and test-benches. These changes include dropping the state-bits corresponding to the registers that have been removed. Also it is very significant that the results of the proposed synthesis are straight-forward to verify, compared to verifying after the most general sequential synthesis. In fact, we prove in Section 4 and confirm experimentally in Section 5 that, in this case, the runtime of the sequential verification is comparable to that of synthesis.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 describes the algorithms. Section 4 discusses verification after the proposed synthesis. Section 5 reports experimental results. Section 6 concludes the paper and outlines future work.

## 2 Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. Terms memory elements, flops, and registers are used interchangeably in this paper.

A node  $n$  has zero or more *fanins*, i.e. nodes that are driving  $n$ , and zero or more *fanouts*, i.e. nodes driven by  $n$ . The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs in combinational optimization and mapping. It is assumed that each node has a unique integer called its *node ID*.

A *fanin (fanout) cone* of node  $n$  is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node  $n$  is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through  $n$ . Informally, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed, the logic in its MFFC can be removed.

*Merging* node  $n$  onto node  $m$  is a structural transformation of a network that transfers the fanouts of  $n$  to  $m$  and removes  $n$  and its MFFC. Merging is often applied to a set of nodes that are proved to be equivalent. In this case, one node is denoted as the *representative* of an equivalence class, and all other nodes of the class are merged onto the representative. The representative can be any node if its fanin cone does not contain any other node of the same class. In this work, the representative is the node of the class that appears first in a topological order.

*SAT sweeping* is a technique for detecting and merging equivalent nodes in a combinational network [15][14][20][21]. SAT sweeping is based on a combination of simulation and Boolean satisfiability. Random simulation is used to divide the nodes into candidate equivalence classes. Next, each pair of nodes in the classes is considered in a topological order. A SAT solver is invoked to prove or disprove the equivalence. If the equivalence is disproved, the counter-example is used to simulate the circuit, which may result in disproving other candidate equivalences. SAT sweeping is used as a robust combinational equivalence checking technique and as a building block in sequential synthesis.

*BMC* stands for bounded model checking. BMC uses Boolean satisfiability to prove a property true for all states reachable from the initial state in a fixed number of transitions (known as *BMC depth*). In the context of equivalence checking, BMC checks pairwise equivalence of the outputs of two circuits under verification.

## 3 Sequential Synthesis and Verification

This section gives an overview of the proposed synthesis algorithms and their use in sequential verification.

### 3.1 Register sweep

Structural register sweep iterates the procedure in Figure 3.1 as long as there is a reduction in the number of registers.

This procedure assumes that all registers have the constant-0 initial state. If some registers have a don't-care initial state, they are transformed by adding a new PIs and a MUX controlled by a special register that produces 0 in the first frame and 1 afterwards. If some registers have a constant-1 initial state, they are

transformed by adding a pair of inverters at the output of the register and retiming the register forward over the first inverter.

Detection of stuck-at-constant registers using ternary simulation is based on [5]. This algorithm assigns the initial values to the registers and simulates the circuit using x-valued primary inputs. The ternary states reached at the registers are collected. Simulation stops when a new ternary state is equal to a previously seen ternary state. At this point, if some register has the same constant value in every reachable ternary state, this register is declared stuck-at-constant.

```

aig runStructuralRegisterSweep(aig N)
{
    // start the set of equivalent register pairs
    set of node subsets Classes =  $\emptyset$ ;

    // detect registers with combinationally-equivalent inputs
    for each register r1 in aig N
        if (there is register r2 in aig N with the same driver as r1)
            Classes = Classes  $\cup$  {r1, r2};

    // detect registers that are stuck-at-constant
    runTernarySimulation(N);
    for each register r1 in aig N
        if (register r1 is stuck-at-constant c)
            Classes = Classes  $\cup$  {c, r1};

    // use the equivalences to reconstruct the aig
    aig N1 = mergeEquivalences(N, Classes);
    return N1;
}

```

Figure 3.1. Structural register sweep.

### 3.2 Signal-correspondence

*Signal-correspondence* is a computation of a set of classes of sequentially-equivalent nodes using induction. The classes are  $K$ -step-inductive in the following sense: (base case) they hold true for all inputs in the first  $K-1$  frames starting from the initial state, and (inductive case) if they are assumed to be true in the first  $K-1$  frames starting from any state, they hold in the  $K$ -th frame.

Our implementation of signal-correspondence follows previous work [9][4][24][18]. The pseudo-code is given in Figure 3.2.

```

aig runSignalCorrespondence(aig N, int K)
{
    // detect candidate equivalences using random simulation
    set of node subsets Classes = randomSimulation(N);

    // refine equivalences by BMC from the initial state for depth K-1
    refineClassesUsingBMC(N, K-1, Classes);

    // perform iterative refinement of candidate equivalence classes
    do {
        // do speculative reduction of K-1 uninitialized frames
        network NR = speculativeReduction(N, K-1, Classes);
        // derive SAT solver containing CNF of the reduced frames
        solver S = transformAIGintoCNF(NR);
        // check candidate equivalences in K-th frame
        performSatSweepingWithConstraints(St, Classes);
    }
    while (Classes are refined during SAT sweeping);

    // merge computed equivalences
    aig N1 = mergeEquivalences(N, Classes);
    return N1;
}

```

Figure 3.2. Signal-correspondence using  $K$ -step induction.

It was found that the scalability of signal-correspondence hinges upon the way the candidate equivalences are assumed before they are proved in the  $K$ -th frame of the inductive case. A technique known as *speculative reduction* was pioneered in [24]. A similar approach was proposed and used in [18].

Speculative reduction merges a node onto its representative for all equivalence classes assumed to hold in the first  $K-1$  frames. After merging, the node is not removed but a constraint is added to assert that this node and the representative are equal. Merging facilitates logic reduction in the fanout cone of the node. This can make the downstream merges trivial and the corresponding constraints redundant. Experiments confirm a dramatic decrease in both the size of the AIG and the number of constraints added to the SAT solver. The gain in runtime due to speculative reduction is several orders of magnitude for large designs.

### 3.3 Partitioned register-correspondence

*Register-correspondence* is a special case of signal-correspondence when candidate equivalences are limited to register outputs, as opposed to all nodes in a sequential network. The implementation of register-correspondence is essentially the same as that of signal-correspondence, except that only register outputs are allowed in the candidate equivalence classes.

It was found that most of the runtime of register-correspondence for large designs is spent in Boolean constraint propagation during the satisfiable SAT runs and simulation of the resulting counter-examples. This led to an idea of partitioning, in which the inductive problem is solved by several instances of a SAT solver, without impacting the completeness of equivalences proved. Since partitions can be solved independently, the partitioned approach lends itself naturally to parallelization.

Our contribution to efficient implementation of register-correspondence is the partitioning algorithm in Figure 3.3.1. The partitions found by the algorithm are used in the procedure **performSatSweepingWithConstraints** of Figure 3.2, which does register-correspondence by limiting candidates to register outputs.

```

set of node subsets partitionOutputs( aig  $N$ , parameters  $Params$  )
{
  // for each PO, compute its structural support in terms of PIs
  set of node subsets  $Supps = \mathbf{findStructuralSupps}(N)$ ;

  // start the output partitions
  set of node subsets  $Parts = \emptyset$ ;

  // add each PO to one of the partitions
  for each PO  $n$  of aig  $N$  in a decreasing order of support sizes {
    node subset  $p = \mathbf{findMinCostPartition}(n, Parts, Params)$ ;
    if ( $p \neq \text{NONE}$ )
       $p = p \cup \{n\}$ ;
    else
       $Parts = Parts \cup \{\{n\}\}$ ;
  }

  // merge small partitions
  compactPartitions(  $Parts, Params$  );
  return  $Parts$ ;
}

```

Figure 3.3.1. Output-partitioning for induction.

The partitioning algorithm in Figure 3.3.1 takes an AIG and a set of parameters, such as bounds on the partition size. The structural supports for all primary outputs are computed in one sweep over the network. The outputs are sorted by support size. The outputs are added to the partitions in the decreasing order of their support sizes. The procedure that determines the partition, to which an output is added, considers the cost of adding the output

to each of the partitions. The cost is defined as a linear combination of attraction (proportional to the number of common variables) and repulsion (proportional to the number of new variables introduced in the partition if the given output is added). The coefficients of the linear combination were found experimentally. Besides the cost considerations, the partitions are not allowed to grow above a given limit. If the best cost of adding an output exceeds another limit, NONE is returned. In this case, the calling procedure starts a new partition. In the end, some partitions are merged if their sizes are below the given minimum.

The key insight allowing for efficient implementation of partitioned register-correspondence is that it is enough to consider one time-frame of the circuit, while signal-correspondence requires at least two time-frames. The partitioning algorithm is fast, which allows repartitioning in each refinement iteration. This leads to improved performance, compared to partitioning upfront and reusing the partitions across multiple iterations. The following observations are used when creating partitions:

- Nodes in a candidate class are added to the same partition.
- Constant candidates can be added to any partition.
- Candidates are merged at the PIs and proved at the POs.
- After solving all partitions, the classes are refined.

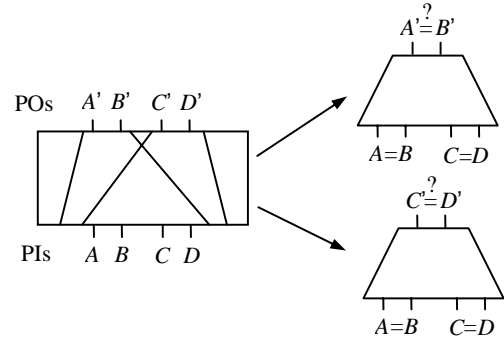


Figure 3.3.2. Illustration of output partitioning for induction.

An example in Figure 3.3.2 illustrates partitioning for induction with two candidate equivalence classes of registers,  $\{A, B\}$  and  $\{C, D\}$ , which are added to different partitions. Note that the equivalences  $A = B$  and  $C = D$  are assumed in both partitions.

### 3.4 Sequential equivalence checking

The pseudo-code in Figure 3.4 shows the sequence of transformations applied by the integrated SEC command in ABC (command “dsec”). The procedure stops as soon as the sequential miter is reduced to a constant, or a counter-example is found. Return shortcuts, which are not shown in the pseudo-code, can take place after any command, for example, after register sweeping or after signal-correspondence. Names of the corresponding stand-alone commands in ABC are given to the left of each procedure call in Figure 3.4. Note that a major part of sequential equivalence checking is the synthesis operations discussed in the first part of this section.

## 4 Scalability of sequential verification

This section present the main verification result of this report related to the scalability of sequential verification after applying the proposed sequential synthesis techniques.

**Theorem.** Let  $N$  be a sequential circuit with a given initial state. Let some signals in  $N$  be proven sequentially-equivalent using  $K$ -step induction. Moreover, let the equivalent signals be merged by

replacing each signal by the representative of its equivalence class. This resulting circuit  $N'$  is sequentially-equivalent to  $N$ . It is assumed that the logic of  $N'$  is not further restructured after merging sequentially-equivalent nodes. Let  $M$  be the sequential miter comparing the POs of the circuits before and after synthesis:  $M = N \oplus N'$ . Then the POs of  $M$  asserted at 0 are inductive invariants provable using  $K$ -step induction, where  $K$  is the same as used in the sequential synthesis.

An important requirement of the above theorem is that there is no further restructuring of the circuit after sequential synthesis. If a restructuring is performed, equivalent points common to both networks may be lost, resulting in the loss of inductiveness [12]. It should be noted that when SEC shown in Figure 3.4 is used to verify the results of the proposed synthesis, the intermediate retiming is disabled (command “dsec -r”). The only logic restructuring command (AIG rewriting) is present inside the  $K$ -loop. This command is never reached if only simple induction ( $K=1$ ) is used in synthesis. In this case, the equivalence is proven by register-correspondence or signal-correspondence with  $K=1$ .

```

sequentialEquivalenceChecker( network N1, network N2 )
{
    // convert networks to AIG; pair PIs/POs by name;
    // transform registers to have constant-0 initial state
    aig M = createSequentialMiter( N1, N2 ); // command “miter -c”

    // remove logic that does not fanout into POs
    runSequentialSweep( M ); // command “scl”

    // remove stuck-at and combinational-equivalent registers
    runStructuralRegisterSweep( M ); // command “scl -l”

    // move all registers forward and compute new initial state;
    // this command can be disabled by switch “-r”, e.g. “dsec -r”
    runForwardRetiming( M ); // command “retime -M 1”

    // merge sequential equivalent registers
    // (this completely solves SEC if only retiming was performed)
    runPartitionedRegisterCorrespondence( M ); // com. “lcorr”

    // merge comb. equivalence before trying signal-correspondence
    runCombinationalSatSweeping( M ); // command “fraig”

    for ( K = 1; K ≤ 64; K = K * 2 ) {
        // merge sequential equivalences by K-step induction
        runSignalCorrespondence( M, K ); // command “ssw -K”

        // minimize and restructure combinational logic
        runAIGrewriting( M ); // command “drw”

        // move registers forward after logic restructuring
        runForwardRetiming( M ); // command “retime -M 1”

        // target satisfiable SAT instances
        runSequentialAIGsimulation( M ); // command
    }

    // if miter is still unsolved, save it for future research
    dumpSequentialMiter( M ); // command “write_aiger”
}

```

Figure 3.4. Integrated SEC in ABC.

Finally, we note another synergy between sequential synthesis and sequential verification, namely the use of retiming. It was found experimentally that applying the most-forward retiming to the sequential miter before  $K$ -step induction often leads to substantial speed-ups - but only if some form of retiming was applied during synthesis! In the synthesis experiments of this report (Section 5), retiming was not used and it was found helpful to skip retiming when verifying the results of this synthesis.

The intuition behind this is the following: As a result of merging nodes during synthesis, structural changes in the circuit

allow registers to travel to different locations after the most-forward retiming. When it happens, alignment of the register locations across the original and the final circuit is lost.

On the other hand, if retiming is used as part of synthesis, then applying most-forward retiming during verification facilitates alignment of the registers in both copies of the design. This alignment is complete if retiming was the only transformation during synthesis. In this case, register-correspondence using simple induction is enough to solve the verification problem [12]. But even if some logic restructuring was done before or after retiming, the most-forward retiming during verification tends to increase the number of matching register locations. This tends to speed up verification and solve difficult verification instances, which were previously not inductive.

## 5 Experimental results

The synthesis and verification algorithms are implemented in ABC [3] as commands *scl*, *lcorr*, *ssw* and *dsec*. The SAT solver used is a modified version of MiniSat-C\_v1.14.1 [8]. The experiments targeting FPGA mapping into 6-input LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The resulting networks are verified by SEC described in this report.

The academic benchmarks used for the experiments are 25 of the largest benchmarks from the ITC’97, ISCAS’89, and IWLS’05 suites [10]. The industrial benchmarks are 50 industrial circuits ranging in size from 100 to 20K registers. For industrial benchmarks with multiple clock domains, optimization was applied only to the domain with the largest number of registers. An additional experiment was run using several IBM benchmarks.

The following ABC commands were included in the scripts used to collect the experimental results:

- *resyn* is a logic synthesis script that performs 5 iterations of AIG rewriting [23].
- *resyn2* is a logic synthesis script that performs 10 iterations of AIG rewriting that are more diverse than those of *resyn*.
- *choice* is a logic synthesis script that allows for accumulation of structural choices [6]; *choice* runs *resyn* followed by *resyn2* and collects three snapshots of the network: the original, the final, and the intermediate one saved after *resyn*
- *if* is a structural FPGA mapper with priority cuts [22], fine-tuned area recovery, and the capacity to map over a subject graph with structural choices [6] (the mapper computes and stores at most 8 6-input priority cuts at each node; it performs five iterations of area recovery, three with area flow and two with exact local area)
- *scl* is a structural register sweep (Section 3.1).
- *lcorr* is a partitioned register-correspondence computation using simple induction (Section 3.3).
- *ssw* is a signal-correspondence computation using  $K$ -step induction (Section 3.2)
- *dsec* is a sequential equivalence checker (Section 3.4)

Tables 1-4 shows results for three experimental runs:

- Section “Baseline” corresponds to a typical run of high-effort technology-independent synthesis and technology mapping with structural choices for FPGA architectures with 6-LUTs (*choice*; *if*; *choice*; *if*; *choice*; *if*)
- Section “Reg Corr” corresponds to register sweep and partitioned register-correspondence (*scl*; *lcorr*) followed by the baseline synthesis and mapping.
- Section “Sig Corr” corresponds to register sweep, partitioned register-correspondence, and signal-correspondence (*scl*; *lcorr*; *ssw*) followed by the baseline synthesis and mapping.

Tables 1-2 show statistics for the academic benchmarks: the number of primary inputs (column “PIs”), primary outputs (column “POs”), and registers (column “Reg”). The area in the tables is calculated as the number of 6-LUTs (columns “LUT”) and delay is measured as the depth of the 6-LUT network (columns “Lev”). The ratios in the tables are the ratios of geometric averages of the values in the columns.

Tables 3-4 show the summary of results for the set of 50 industrial benchmarks. In Table 3, columns “Baseline”, “Reg Corr”, and “Sig Corr” show geometric averages of the corresponding parameters. Columns “Ratio” show the ratios of the columns to the left against column “Baseline”. The rows show the number of registers, area (the number of 6-LUTs), and delay (the depth of 6-LUT network). In Table 4, columns correspond to transformations. The row “Geomean” shows the average runtimes taken by each transformation. Row “Ratio” shows the ratios of the runtimes to the total runtime of synthesis and verification.

### Synthesis results

A substantial reduction (about 33%) over the baseline case is achieved on academic benchmarks (Table 1) in both registers and area while the delay is reduced by 14%. Such large reductions on these benchmarks may be the result of unrealistic initial states that were possibly reset to the all-0 state after the original initial state was lost. In this case, the proposed sequential synthesis removes the logic that is not exercised on the reachable states.

The reduction achieved for the industrial benchmarks (Table 3) is 32% in registers and 15% in area while preserving the delay. The geometric averages of synthesis and verification runtime on this set are close to 50 sec and 25 sec, respectively.

An additional experiment was run on a set of 9 industrial benchmarks contributed by our collaborators at IBM. The results are shown in Table 5, which has the same notation as Table 1. On this benchmark set, the reduction in registers and LUTs is 14% and 9%, respectively. The total runtime of synthesis and verification followed by combinational synthesis and mapping was less than one minute for all 9 benchmarks from this set.

### Verification results

The verification runtimes and runtime ratios are shown in Tables 2 and 4 in columns “SEC”.

It is interesting to note that, for academic benchmarks, the verification runtime is larger than that of sequential synthesis, possibly because larger synthesis reductions lead to larger structural gaps that should be bridged by Boolean reasoning during the iterative refinement. For the industrial benchmarks, the verification takes less time than sequential synthesis. This is likely because verification uses the partitioned implementation of register-correspondence to quickly reduce the size of the sequential miter composed of two similar copies of the same design, leaving less if any work to do for slower signal-correspondence. Meanwhile, synthesis applies signal-correspondence to a single copy of the design reduced only by 30% on average after partitioned register-correspondence.

Finally, we mention that sequential verification capabilities of ABC were applied to several sets of industrial problems ranging in size from less than a hundred to several thousand registers. Most of these problems were solved successfully within several minutes while, on some of them, the verification engine gave up after scaling  $K$ -step induction to a predefined limit. The detailed results are not included in this report because the type of synthesis is not known and therefore these results are not relevant to the case study of synergy between synthesis and verification.

## 6 Conclusions and future work

This report presents a case-study of a sequential synthesis and its counterpart, a scalable approach to sequential verification that can efficiently validate the results of the proposed synthesis. The presented algorithms have the following salient features:

- efficient implementation using partitioned inductive solving.
- substantial savings in registers and area without increasing the delay.
- minimum modification to state encoding, scan chains, and functional test vectors after synthesis.
- scalable sequential verification by an inductive prover comparable to that used for synthesis.

The experimental results on both academic and industrial benchmarks confirm the practicality of the proposed synthesis and demonstrate affordable runtimes of the unbounded sequential equivalence checking after sequential synthesis.

Future work in this area will include:

- Tuning the inductive prover for scalability (for example, using unique-state constraints).
- Developing new sequential engines (interpolation [19], synthesizing equivalence for induction, etc).
- Extending the proposed form of sequential synthesis to include (a) on-the-fly retiming [1], (b) logic restructuring using unreachable states as external don’t-cares, (c) iterative processing similar to that of combinational synthesis [23].

## Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, and the California Micro Program with industrial sponsors Actel, Altera, Calypto, Magma, Synopsys, and Synplicity.

The authors are indebted to Stephen Jang of Xilinx for his masterful experimental evaluation of the proposed algorithms.

## References

- [1] J. Baumgartner and A. Kuehlmann, “Min-area retiming on flexible circuit structures”, *Proc. ICCAD’01*, pp. 176-182.
- [2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. “Scalable sequential equivalence checking across arbitrary design transformations”. *Proc. ICCD’06*.
- [3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Release 70930. <http://www-cad.eecs.berkeley.edu/~alanmi/abc>
- [4] P. Bjesse and K. Claessen. “SAT-based verification without state space traversal”. *Proc. FMCAD’00*. LNCS, Vol. 1954, pp. 372-389.
- [5] P. Bjesse and J. Kukula, “Automatic generalized phase abstraction for formal verification”, *Proc. ICCAD’06*, pp. 1076-1082.
- [6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping”, *Proc. ICCAD’05*, pp. 519-526.
- [7] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”, *IEEE Trans. CAD*, vol. 13(1), January 1994, pp. 1-12.
- [8] N. Een and N. Sörensson, “An extensible SAT-solver”. *SAT ’03*. <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat>
- [9] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities, *IEEE Trans. CAD*, vol. 19(7), July 2000, pp. 814-819.
- [10] IWLS 2005 Benchmarks. <http://iwls.org/iwls2005/benchmarks.html>
- [11] J.-H. Jiang and R. Brayton, “Retiming and resynthesis: A complexity perspective”. *IEEE TCAD*, Vol. 25 (12), Dec. 2006, pp. 2674-2686.
- [12] J.-H. Jiang and W.-L. Hung, “Inductive equivalence checking under retiming and resynthesis”, *Proc. ICCAD’07*.

- [13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *IEEE TCAD*, Vol. 21(12), Dec 2002, pp. 1377-1394.
- [14] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking". *Proc. ICCAD '04*, pp. 50-57.
- [15] F. Lu, L. Wang, K. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," *Proc. DAC '03*, pp. 668-673.
- [16] F. Lu, M. K. Iyer, G. Parthasarathy, K.-T. Cheng, and K.C. Chen. "An efficient sequential SAT solver with improved search strategies". *Proc. DATE '05*, pp. 1102-1107.
- [17] F. Lu and K.-T. Cheng. "Sequential equivalence checking based on K-th invariants and circuit SAT solving". *Proc. HLDVT '05*, pp. 45-51.
- [18] F. Lu and T. Cheng. "IChecker: An efficient checker for inductive invariants". *Proc. HLDVT '06*, pp. 176-180.
- [19] K. L. McMillan. "Interpolation and SAT-Based model checking". *Proc. CAV '03*, pp. 1-13
- [20] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report*, EECS Dept., U. C. Berkeley, March 2005.
- [21] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843
- [22] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*.
- [23] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.
- [24] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. "Exploiting suspected redundancy without proving it". *Proc. DAC '05*.
- [25] K. Ng, M. R. Prasad, R. Mukherjee and J. Jain, "Solving the latch mapping problem in an industrial setting," *Proc. DAC '03*, pp. 442-447.
- [26] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

Table 1. Register count, area, and delay after the proposed synthesis for academic benchmarks.

Example	Statistics		Baseline			Reg Corr			Sig Corr		
	PI	PO	LUT	Reg	Lev	LUT	Reg	Lev	LUT	Reg	Lev
<b>b14</b>	32	54	1144	245	13	1090	215	13	1122	215	13
<b>b15</b>	36	70	2027	449	14	1930	415	15	1940	415	15
<b>b17</b>	37	97	6086	1415	18	2356	611	15	2327	604	14
<b>b20</b>	32	22	2454	490	14	2400	429	14	2398	429	14
<b>b21</b>	32	22	2439	490	15	2395	429	15	2324	429	15
<b>b22</b>	32	22	3720	735	14	3602	611	14	3493	611	14
<b>s13207</b>	31	121	655	669	5	255	195	3	252	193	3
<b>s35932</b>	35	320	2720	1728	2	2336	1472	2	2336	1472	2
<b>s38417</b>	28	106	2130	1636	6	1859	1348	6	1848	1345	6
<b>s38584</b>	12	278	2242	1452	6	1199	843	4	1120	784	4
<b>systemcaes</b>	260	129	1851	670	8	1796	670	8	1795	670	8
<b>systemcdes</b>	132	65	405	190	7	419	190	6	476	190	5
<b>tv80</b>	14	32	1787	359	11	1834	350	10	110	64	3
<b>usb_funct</b>	128	121	3095	1746	6	3044	1722	6	2758	1602	6
<b>vga_lcd</b>	89	109	28939	17079	6	26891	16001	6	22274	12882	6
<b>wb_conmax</b>	1130	1416	10303	770	6	9851	770	6	9865	770	6
<b>wb_dma</b>	217	215	1037	563	6	898	521	4	879	490	4
<b>ac97_ctrl</b>	84	48	2798	2199	3	2792	2199	3	2570	1985	3
<b>aes_core</b>	259	129	2951	530	5	2958	527	5	2962	527	5
<b>des_area</b>	240	64	937	128	7	867	64	7	872	64	7
<b>des_perf</b>	234	64	5516	8808	4	5525	8746	4	5518	8746	4
<b>Ethernet</b>	98	115	11939	10544	6	1396	883	6	1357	849	6
<b>i2c</b>	19	14	232	128	3	247	126	3	217	126	3
<b>mem_ctrl</b>	115	152	1916	1083	7	1890	1046	7	1854	1046	5
<b>pci_spoci_ctrl</b>	25	13	238	60	4	238	60	4	41	33	3
<b>Geomean</b>	The		2141	809.9	6.8	1725	610.9	6.33	1405	544.3	5.83
<b>Ratio</b>			<b>1</b>	<b>1</b>	<b>1</b>	<b>0.806</b>	<b>0.754</b>	<b>0.935</b>	<b>0.656</b>	<b>0.672</b>	<b>0.86</b>
<b>Ratio</b>						<b>1</b>	<b>1</b>	<b>1</b>	<b>0.814</b>	<b>0.891</b>	<b>0.92</b>

Table 2. Runtime of the proposed synthesis and verification for academic benchmarks.

Example	Statistics			Runtime Distribution				
	PI	PO	Reg	RegCorr	SigCorr	SEC	Mapping	Total
b14	32	54	245	0.03	0.31	0.63	7.65	8.62
b15	36	70	449	0.34	6.26	11.79	12.24	30.63
b17	37	97	1415	0.67	11.32	19.23	12.46	43.68
b20	32	22	490	0.1	1.79	2.87	15.72	20.48
b21	32	22	490	0.11	2.55	3.96	16.54	23.16
b22	32	22	735	0.13	2.9	6.88	22.66	32.57
s13207	31	121	669	0.06	0.22	0.31	0.46	1.05
s35932	35	320	1728	0.57	1.83	5.43	5.47	13.3
s38417	28	106	1636	1.05	2.65	8.59	6.09	18.38
s38584	12	278	1452	0.7	1.12	4.11	3.36	9.29
Systemcaes	260	129	670	0.07	1.42	2.81	11.17	15.47
Systemcdes	132	65	190	0.01	0.07	0.09	3.7	3.87
tv80	14	32	359	0.22	0.94	1.28	0.46	2.9
usb_funct	128	121	1746	1.95	12.65	29.64	11.36	55.6
vga_lcd	89	109	17079	165.01	570.28	1830.19	78.85	2644.33
wb_conmax	1130	1416	770	0.64	17.08	2.03	45.03	64.78
wb_dma	217	215	563	0.11	0.49	1.29	2.54	4.43
ac97_ctrl	84	48	2199	0.69	14.83	25.31	6.38	47.21
aes_core	259	129	530	0.14	1.25	1.65	27.51	30.55
des_area	240	64	128	0.01	0.13	0.1	5.45	5.69
des_perf	234	64	8808	1.1	81.97	49.44	113.52	246.03
Ethernet	98	115	10544	4.93	3.68	19.66	4.15	32.42
i2c	19	14	128	0.03	0.11	0.19	0.7	1.03
mem_ctrl	115	152	1083	0.96	10.21	11.96	5.4	28.53
pci_spoci_ctrl	25	13	60	0.03	0.09	0.15	0.15	0.42
Geomean			809.9	7.1864	29.846	81.5836	16.7608	135.3768
Ratio				<b>0.053</b>	<b>0.221</b>	<b>0.603</b>	<b>0.123</b>	<b>1.000</b>

Table 3. Register count, area, and delay after the proposed synthesis for industrial benchmarks.

	Baseline	Reg Corr	Ratio	Sig Corr	Ratio
Registers	1583	1134	<b>0.71</b>	1084	<b>0.68</b>
6-LUTs	7472	6751.5	<b>0.90</b>	6360	<b>0.85</b>
Depth	8.5	8.5	<b>1.00</b>	8.5	<b>1.00</b>

Table 4. Runtime of the proposed synthesis and verification for industrial benchmarks.

	Reg Corr	Sig Corr	SEC	Syn & Map	Total
Geomean	3.31	74.28	41.10	40.60	159.30
Ratio	<b>0.02</b>	<b>0.47</b>	<b>0.26</b>	<b>0.25</b>	<b>1.00</b>

Table 5. Register count, area, and delay after the proposed synthesis for a set of IBM benchmarks.

Example	Statistics		Baseline			Reg Corr			Sig Corr		
	PI	PO	LUT	Reg	Lev	LUT	Reg	Lev	LUT	Reg	Lev
<b>bs1</b>	306	265	1168	720	4	1116	678	4	1102	667	4
<b>bs2</b>	362	363	1630	816	5	1548	748	6	1538	742	6
<b>bs4</b>	111	202	734	284	5	692	255	5	691	254	5
<b>bs5</b>	66	12	183	100	3	164	93	3	164	93	3
<b>bs6</b>	193	74	335	172	6	312	151	6	286	142	6
<b>bs7</b>	182	116	1031	442	4	1042	428	4	1025	425	4
<b>bs8</b>	202	23	1601	365	5	1330	207	5	1306	205	5
<b>bs9</b>	228	124	1148	391	3	1045	353	3	1037	353	3
<b>Ratio</b>			<b>1.00</b>	<b>1.00</b>	<b>1.00</b>	<b>0.93</b>	<b>0.88</b>	<b>1.02</b>	<b>0.91</b>	<b>0.86</b>	<b>1.02</b>