

Temporal Decomposition for Logic Optimization

Nathan Kitchen¹

Andreas Kuehlmann^{1,2}

¹ University of California at Berkeley, CA, USA

² Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

*Traditional approaches for sequential logic optimization include (1) explicit state-based techniques such as state minimization, (2) structural techniques such as retiming, and (3) methods that exploit sequential don't-cares derived from unreachable states. These approaches optimize a logic circuit as a single component with a single input/output behavior. In this paper we present a novel concept for sequential optimization referred to as **temporal decomposition**, which distinguishes the logic that initializes the circuit from the logic needed for the behavior after startup. This work was motivated by a recent observation made for bounded property verification: There is a substantial optimization potential for transition relations when the first execution steps are applied as satisfiability don't-cares. This result suggests that current designs include circuitry that is only used during the first few clock periods after reset and could be discarded or disabled after startup. In this paper we describe how temporal decomposition could be applied to treat the logic for startup separately from the remaining circuitry and discuss multiple alternatives to exploit this for an improved implementation.*

1. Introduction

Past research into sequential circuit optimization has yielded a variety of approaches. Classical state minimization techniques [1] aim at lowering the complexity of the state-transition graph by reducing the number of states. Retiming [2, 3] moves the registers and latches of a circuit to reduce the maximum delay through the combinational logic and thus improve the circuit performance. A similar result is achieved by clock skew scheduling [4] which adjusts the timing of the register clocking to balance the delays of the combinational logic. Reachability-based methods (e.g. [5]) derive don't-cares from unreachable states and use them to minimize the logic beyond what combinational approaches allow. While the ways in which these approaches seek to reduce a circuit's implementation cost differ, they have in common that they treat the circuit as a single component whose input/output (I/O) behavior would be preserved from the initial state throughout its execution.

In this paper, we present a novel approach to optimizing sequential synchronous circuits which we refer to as *temporal decomposition*. The idea of this new concept is based on the observation that the logic needed for startup is used only once and could be “discarded” or “deactivated” once the startup sequence has been executed. For temporal decomposition, the circuit's I/O behavior is divided into two parts: (1) the *startup behavior* which includes the initial circuit execution for a fixed number of clock cycles and (2) the *recurrent behavior* which includes the execution from the states reached after the startup period. The distinction between

startup and recurrent behavior was first suggested by the results in a recent publication on bounded model checking (BMC) [6]. In that work, satisfiability don't-cares from the first clock cycles are applied to simplify the transition relation for property checking. The experiments showed an average decrease of 38% in the size of the circuit graphs implementing the transition relations after only a few cycles. This result implies that a significant fraction of the logic in a circuit is needed only to initialize it in the first few cycles after reset. While surprising, this result is intuitively explicable: In the RTL-based design practice, circuit designers commonly think about initialization as a functional requirement, but do not necessarily consider the effort to implement it.

For synthesis, the separation of the startup behavior from the recurrent behavior can be exploited for optimizing both parts independently and thus implementing them more efficiently. For example, by using modern rapidly reconfigurable fabrics, one can first program and execute the optimized startup logic, and after startup, reprogram the platform for the recurrent behavior. In an ASIC-based implementation one could implement both components and use means of power management such as voltage islands [7], to turn them on and off.

In this paper, we present several schemes for implementing the two components, as well as the outcomes of some preliminary experiments. To the best of our knowledge, this paper is the first to introduce the concept of temporally dividing the circuit execution into two (or more) sequences which are optimized separately. Our initial results based on a don't-care optimization algorithm for the recurrent behavior are encouraging and demonstrate a significant reduction potential for some of the benchmark circuits. We believe that, as this research continues, more powerful algorithms can further improve the results and lead to an attractive implementation scheme that takes advantage of modern fabrics such as rapidly reprogrammable platforms.

This paper is structured as follows: Section 2 introduces our notation and explains the theoretical basis for temporal decomposition. Section 3 presents several approaches for implementing temporal decomposition. In Section 4, we describe a method for computing the recurrent component in the decomposition. Section 5 presents our experimental results. Finally, Section 6 contains conclusions and future work.

2. Preliminaries

In this section we introduce the theoretical background for the temporal decomposition technique described in this paper, adapting the terminology and notation of [6, 8]. Let x denote the set of primary inputs, y be the set of primary outputs, s be the current state, and s' be the next state. Let δ be the vector function that computes the next-state bits, so $\delta(x, s) = s'$. Let λ be the vector

function that computes the primary outputs, so $\lambda(x, s) = y$. The *transition relation* $T(s, s', x, y)$ is the predicate that holds when its arguments are consistent with the next-state and output functions:

$$T(s, s', x, y) = s' \Leftrightarrow \delta(x, s) \wedge y \Leftrightarrow \lambda(x, s) \quad (1)$$

For simplicity, we often abbreviate $T(s, s', x, y)$ as $T(s, s')$. Figure 1 illustrates the given notation for a finite-state-machine (FSM) structure.

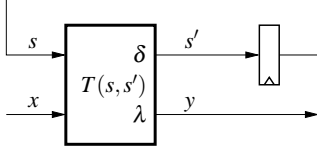


Figure 1: Illustration of the notation used, showing a circuit as an FSM with primary inputs x , primary outputs y , current state variables s , next-state variables s' , next-state function (vector) $s' = \delta(x, s)$, and output function (vector) $y = \lambda(x, s)$. The FSM implements transition relation $T(s, s')$.

In general, the range of the next-state function δ does not include all states; some states are not reachable by transitions from any other states. Other states are reachable only by transitions from these unreachable states. These transitively unreachable states can be exploited for logic optimization. In order to describe the relation between reachability and optimization formally, we define the property of *k-reachability*. A state s is *k-reachable* if it can be reached from some state in k transitions. The predicate $R_k(s)$, which holds when s is *k-reachable*, is defined in terms of the transition relation T :

$$R_0(s_0) = 1 \quad (2)$$

$$R_k(s_k) = \exists s_{k-1}, x, y R_{k-1}(s_{k-1}) \wedge T(s_{k-1}, s_k, x, y) \quad (3)$$

Note that the number of *k-reachable* states is non-increasing in k , since $R_k(s) \Rightarrow R_{k-1}(s)$, but $R_{k-1}(s) \not\Rightarrow R_k(s)$. Note also that *k-reachability*, as defined here, is not the same property checked during standard forward reachability analysis in logic verification. In the standard analysis, reachability is checked from a set of designated initial states. Here, we consider a state reachable if it can be reached from any state, not just one of the initial states.

To represent logic optimization formally, we use the *generalized cofactor*, which we denote as $A(s)|_{B(s)}$. For two predicates A and B , the value of the generalized cofactor is an incompletely specified predicate, defined as follows:

$$A(s)|_{B(s)} = \begin{cases} A(s) & \text{if } B(s) = 1 \\ \text{don't care} & \text{otherwise} \end{cases} \quad (4)$$

By *don't care* we mean that an optimization procedure can pick any value for $A(s)$, providing a means for improving its circuit implementation.

We now describe how *k-reachability* leads to optimization potential. The same behavior implemented by a simple FSM can be implemented by an unrolled version of it, shown in Figure 2, if the outputs are multiplexed appropriately. (Primary inputs and outputs are omitted from the figure for simplicity.) The different stages of the unrolled circuit are used during different cycles of execution:

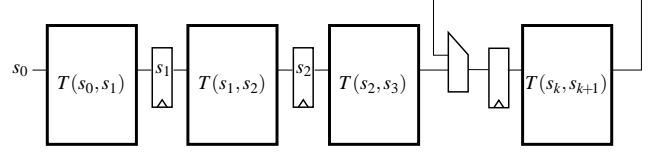


Figure 2: Illustration of unrolled circuit. For simplicity, primary inputs and outputs are omitted. s_0 is an arbitrary starting state.

Each of the first stages is used for a single cycle after reset, and the last stage is used for the remainder of execution. Although the starting state s_0 is arbitrary, the values of s_1 , s_2 , and so on, are not; they must be *k-reachable*. For example, the input state of the second stage, s_1 , is 1-reachable, because it is reached by a transition in the first stage. As a result, the second stage does not need to implement transitions from states that are not 1-reachable. It may be replaced by optimized logic that implements a simplified transition relation $\tilde{T}_2(s_1, s_2) = T(s_1, s_2)|_{R_1(s_1)}$ without altering the overall behavior. Likewise, each stage k may be replaced by logic implementing an optimized transition relation $\tilde{T}_k(s_{k-1}, s_k)$:

$$\tilde{T}_k(s_{k-1}, s_k) = T(s_{k-1}, s_k)|_{R_{k-1}(s_{k-1})} \quad (5)$$

The optimization uses *k-unreachable* states as *satisfiability don't-cares* (SDCs). Since the number of *k-reachable* states is non-increasing in k , the number of *k-unreachable* states is non-decreasing, and the optimization potential of each stage is greater than the previous one (or at least no less). The surprising fact illustrated by the results of [6] is how much optimization is possible after just a few clock cycles.

3. Optimization by Temporal Decomposition

In the previous section we showed that the transition relation describing a circuit's behavior in a particular clock cycle can be simplified using SDCs derived from previous cycles. The simplification cannot be applied in every time frame; it is valid only after the time frames that give rise to the SDCs. Therefore, the simplification potential is realized by *temporal decomposition* of the circuit: Its behavior is divided into two sequences, and each sequence is implemented with a separate component.

3.1. Decomposition

The *startup behavior* consists of a fixed number of clock cycles after reset. The number of startup cycles, denoted K , must be chosen well in order to realize the benefits of temporal decomposition, as we discuss later in this section. The remainder of execution constitutes the *recurrent behavior*, in which simplification using the SDCs from the startup cycles is valid.

Figure 3 illustrates the decomposition of a circuit into the components that implement the two behaviors. The *startup component* C_S is used only for the K cycles of the startup. At the end of the startup sequence, C_S passes the state s_K to the *recurrent component* C_R , which continues with the long-term execution, while C_S is disabled. Disabling C_S is crucial to obtain real improvement from temporal decomposition. If C_S is allowed to continue running after the startup cycles, it will consume resources needlessly and negate the benefits of the decomposition.

Each component is optimized for the behavior it implements. The recurrent component implements the simplified transition relation $\tilde{T}_K(s, s')$ described in Section 2. The startup component can

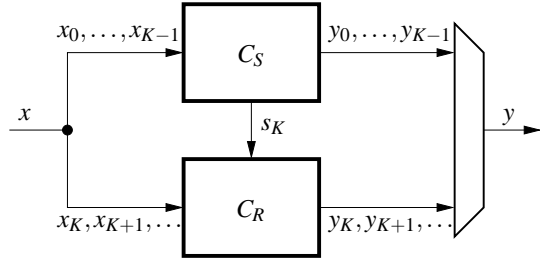


Figure 3: Conceptual view of temporal decomposition of a circuit with primary inputs x and primary outputs y into an startup component C_S and a recurrent component C_R .

also be simplified using SDCs if the reset state includes known values for some of the registers. (This simplification is not described by the simplified relations \tilde{T}_k .)

The length of the startup sequence, K , is a critical parameter in temporal decomposition. If K is too small, the startup frames provide few SDCs to minimize C_R . On the other hand, the implementation cost of C_S increases with K . The total cost for various values of K depends on the particular circuit, the cost metric of interest, and the method of implementation.

An example of how the implementation cost may vary with K is sketched in Figure 4. The plot illustrates qualitatively the areas of the startup and recurrent components of a hypothetical circuit, as well as the total area for two types of implementation. In an implementation where the components are separated spatially, the total area of the temporally decomposed circuit exceeds the area of the original circuit. In contrast, an implementation that separates the components temporally, such as a reconfigurable fabric, requires less area if K is chosen well.

In addition to the choice of K , the benefits of temporal decomposition depend on the kind of circuit being decomposed. Circuits whose state-transition structure includes few k -unreachable states have less potential for optimization by temporal decomposition. In particular, datapath circuitry has few unreachable states—if any—so little gain can be expected from decomposing it, in contrast with control circuitry.

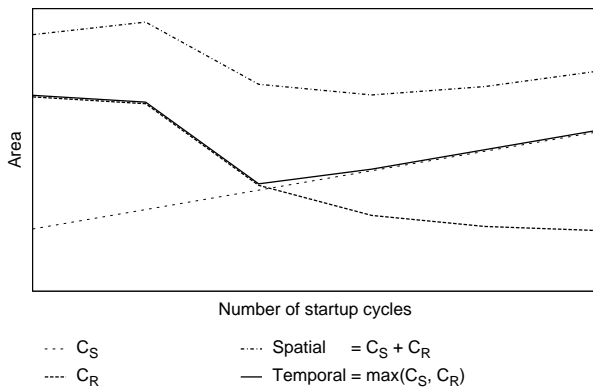


Figure 4: Sketch of tradeoff between number of cycles in startup sequence and costs of temporal decomposition.

3.2. Implementation Approaches

The components C_S and C_R may be implemented in a variety of ways. We discuss several alternatives here; our list is by no means exhaustive.

Next-generation programmable logic: An implementation of temporal decomposition using a rapidly reconfigurable fabric reduces area overhead by using separate configurations for C_S and C_R . The configuration for C_S is loaded initially. After the startup cycles, a counter triggers the loading of the C_R configuration.

The capability for rapid configuration switching—that is, a switching time on the order of a few cycles—is critical to make this alternative viable. Many fabrics currently in use require hundreds of cycles or more for reconfiguration, and are thus unsuited to this implementation.

Voltage islands: In this implementation approach, C_S and C_R are separate functional blocks, each with its own voltage supply. After the K cycles of startup, C_S 's supply is turned off. The outputs from the two components are routed through a multiplexor. This scheme imposes overhead in the form of extra area and may require added buffering to handle the extra input loading. Another consideration is the effect on routing. In cases where temporal decomposition allows aggressive optimization of C_R , the costs could be offset by lower total power consumption.

Slow clocking: A third method focuses on gates whose outputs are not functionally equivalent under all inputs, but whose functional differences coincide with the SDCs from the first cycles. Figure 5 illustrates the concept for a pair of such gates g_R and g_S . After the startup cycles, their output functions are indistinguishable, so the fanouts of g_S can be switched over to g_R . g_S is effectively left dangling; its output function becomes irrelevant. This fact can be exploited by using a lower-power gate with greater delay for g_S . With added frequency-control circuitry, the clock period can be lengthened during startup when g_S 's output is used, then shortened after the switch to g_R . Timing violations on g_S will have no effect on the circuit's behavior. The effect of the slower cycles on the overall execution time is negligible, because startup takes just a few cycles.

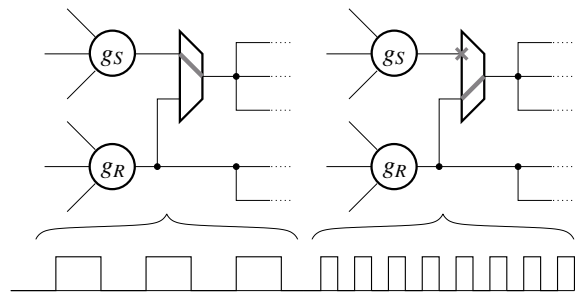


Figure 5: Illustration of slow clocking.

4. Don't-Care Optimization

Having introduced the concept of temporal decomposition and suggested approaches to implementation, we now describe a method for optimizing the recurrent component C_R . Our method was inspired by [9]. Similarly to that work, we use Boolean satisfiability (SAT) to find local don't-care cubes, and we control computational complexity by limiting the number of literals in the cubes.

In contrast with [9], we do not consider observability nor use windowing.

4.1. Overall Flow

The pseudocode for our method of optimizing the recurrent component C_R is shown in Algorithm 1. Figure 6 illustrates some of the concepts and notation. We assume that a Boolean network N is composed of complex nodes $\{u\}$ which we want to minimize using local SDCs. COMPUTE- C_R takes the following parameters: N , a Boolean network; k_{\max} , the maximum number of frames to unroll; and n_{lit} , the limit on the number of literals in don't-care cubes. The network is unrolled into frames $N_1, \dots, N_{k_{\max}}$, and a simplified network \tilde{N}_k is constructed from each frame. The simplification of a frame is performed by minimizing each node with respect to SDCs obtained from the previous frames by the COMPUTESDCS routine (described in Section 4.2). Each \tilde{N}_k is considered as a candidate for C_R .

Algorithm 1 COMPUTE- $C_R(N, k_{\max}, n_{\text{lit}})$

```

1: CLUSTERNODES( $N$ )
2:  $\{N_1, \dots, N_{k_{\max}}\} \leftarrow \text{UNROLL}(N, k_{\max})$ 
3: CONSTRUCTAIG( $\{N_k\}$ )
4: for each frame  $N_k$  do
5:    $\tilde{N}_k \leftarrow \text{NEWNETWORK}()$ 
6:   for each node  $u$  in  $N_k$  do
7:      $D_u \leftarrow \text{COMPUTESDCS}(u, n_{\text{lit}})$ 
8:      $\tilde{u} \leftarrow \text{MINIMIZE}(u, D_u)$ 
9:     Add  $\tilde{u}$  to  $\tilde{N}_k$ 
10:  $K \leftarrow \arg \min_k \{\text{COST}(\tilde{N}_k) \mid \text{COST}(\tilde{N}_k) \ll \text{COST}(\tilde{N}_{k-1})\}$ 
11:  $C_R \leftarrow \tilde{N}_K$ 

```

Minimization has little effect on nodes with few inputs, because their functions are already simple. We avoid this problem with a pre-processing step (line 1) that clusters nodes together in order to create larger nodes.

For Boolean reasoning in the don't-care computation, we use AND-INVERTER graphs (AIGs) [6]. AIGs are semi-canonical circuit graphs that employ on-the-fly simplifications such as constant folding and structural hashing to reduce redundancy. They contain three types of vertices: constant 0, inputs, and 2-input AND gates. Inverters are represented compactly by attributes on references to vertices. We construct a single AIG to represent all the unrolled frames (line 3), thus converting the dependences between values in different frames to combinational dependences. In general, each node u maps to several vertices in the AIG. (See Figure 7 for an illustration.)

The MINIMIZE routine (line 8) is a two-level minimization routine similar to Espresso [10].

After the simplified networks \tilde{N}_k are constructed, their costs are compared and one of them is chosen as C_R . The criterion in line 10 selects the last frame with significantly less cost than its predecessors, that is, the last frame before the gains taper off.

The maximum number of literals in each cube, n_{lit} , is a critical parameter. The complexity of COMPUTESDCS increases super-exponentially with n_{lit} , so it is only feasible to consider cubes with few literals. This constraint is less limiting than it seems, because cubes with few literals are more likely to be useful in logic minimization, due to the large number of minterms that they cover. Even for small values of n_{lit} , the number of cubes considered by

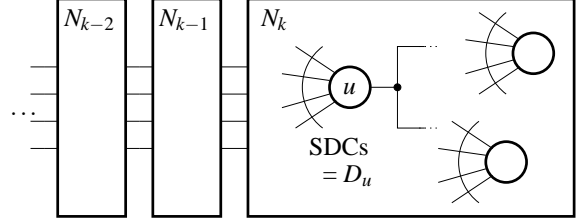


Figure 6: Illustration of unrolled frames N_1, \dots, N_k in computation of C_R . Each node u is minimized using D_u , a set of SDC cubes over its inputs.

COMPUTESDCS is a strong function of u 's fanin. For this reason, we impose a fanin limit during the clustering step. No nodes with fanins above the limit are created.

4.2. Computation of SDC Cubes

The COMPUTESDCS routine is shown in Algorithm 2. To compute SDC cubes for a node in a frame, we enumerate the node's input cubes and check for satisfiability of each one. Only cubes with few literals are enumerated. Satisfiability of cubes is detected in two ways: by occurrence in simulation vectors, and by a technique called SAT sweeping [6]. SAT sweeping efficiently identifies functional equivalences in AIGs by interleaving word-parallel simulation and SAT checks.

Algorithm 2 COMPUTESDCS(u, n_{lit})

```

1:  $D_u \leftarrow \{\}$  // Don't-care set (approx.)
2: for  $i = 1$  to  $n_{\text{lit}}$  do
3:    $V_{\text{cand}} \leftarrow \{\}$  // Vertices for candidate cubes
4:   for each cube  $C$  over  $i$  inputs of  $u$  do
5:     unless  $C$  occurs in simulation vectors of  $u$ 's inputs
6:       or any cube in  $D_u$  subsumes  $C$ 
7:       or  $\exists$  literal  $l_j$  in  $C$  s.t.  $v_{l_j} \equiv \bar{v}_0$  do
8:          $v_C \leftarrow \text{AND}(v_{l_1}, v_{l_2}, \dots)$  for  $l_j$  in  $C$ 
9:         Add  $v_C$  to  $V_{\text{cand}}$ 
10:    SAT-SWEEPING( $V_{\text{cand}} \cup \{v_0\}$ )
11:    for each  $v_C$  in  $V_{\text{cand}}$  do
12:      if  $v_C \equiv v_0$  then // If  $v_C$  is constant 0,
13:        Add  $C$  to  $D_u$  // then the cube is an SDC.
14: return  $D_u$ 

```

Given a node u , COMPUTESDCS enumerates all input cubes of u with no more than n_{lit} literals, in increasing order of literal count (lines 2, 4). The literal-count ordering is not necessary for the correctness of the algorithm, but it improves efficiency by preventing consideration of redundant cubes. For example, suppose that the 2-cube xy is determined to be in the don't-care set D_u . The 3-cube xyz should not be considered for inclusion in D_u , because it is subsumed by xy . Because all the 2-literal DC cubes are added to D_u before the 3-cubes are enumerated, the subsumption is guaranteed to be detected during the check in line 6, and no unnecessary vertices are created in the AIG.

Line 7 handles the case where a literal's value is known to be 1 because the corresponding input was determined to be constant by a previous invocation of SAT-SWEEPING. In this case, an i -cube reduces to an $(i-1)$ -cube. Since all 1- to $(i-1)$ -cubes have already

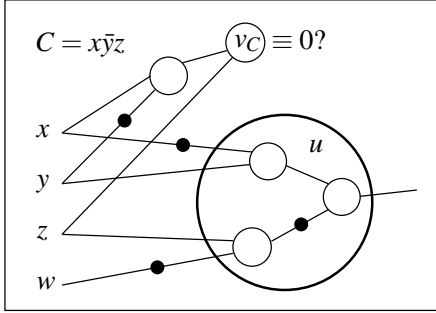


Figure 7: Illustration of AIG vertices for a node u and a local input cube $C = x\bar{y}z$. Edges with inverters are indicated by dots.

been considered in previous iterations of the outer loop, there is no need to check the i -cube.

Every cube C that does not appear in the simulation vectors, is not subsumed by known DC cubes, and does not have constant-valued literals is a candidate for inclusion in D_u . To check the satisfiability of a candidate cube, vertices are added to the AIG to represent it (line 8). Figure 7 illustrates the tree of AND vertices within the AIG that represent a cube C , including the root v_C . SAT sweeping checks each such v_C for equivalence with constant 0 (line 10). Equivalence of a vertex v_C with constant 0 indicates that the literals in C cannot be simultaneously justified to 1 by any choice of primary input values in the current and previous frames. Therefore, C is an SDC. Most vertices are distinguished from 0 early in SAT sweeping by random simulation; the same simulation values are useful for filtering candidate DC cubes, because they tend to include many different minterms in a node’s care set.

5. Experimental Results

We implemented the don’t-care optimization procedure from the previous section in the SIS environment [11]. In this section we present the results of our experiments. Our input circuits included a subset of the ITC’99 benchmarks (available at <http://www.cad.polito.it/tools/itc99.html>) and several modules from the OpenCores repository (available at <http://www.opencores.org>). All circuits were synthesized to optimized gate-level netlists with an industrial logic synthesis tool. We clustered the gates into Boolean nodes, using a node fanin limit of 20. In the SDC computation, we considered cubes with up to 3 literals ($n_{lit} = 3$). For SAT checks, we used MiniSat [12].

Table 1 lists the circuit characteristics and results. Columns 2 and 3 show gate and register counts for the netlists before preprocessing. Columns 4 and 5 give literal counts for the first frame of the unrolling and the frame with the fewest literals after simplification. The ratio between these two values is given in Column 6. The reduction in literal count achieved by using SDCs from previous frames varies from 0 to 26%, with an average of 5%. Note that the reduction is measured relative to the simplified first frame, so that the decrease is due only to SDCs from previous frames. Any gains due to SDCs within frames (that is, combinational SDCs) are obtained in the first frame; they are factored out of the reported reduction results.

For comparison, Table 1 also includes the literal counts achieved by running *script.rugged*, which uses BDDs for image

| Design | Gates | Regs | Fr. 1 | Min. Fr. | Rel. to Fr. 1 | script.rugged | Rel. to Fr. 1 |
|-------------|-------|------|-------|----------|---------------|---------------|---------------|
| [ITC’99] | | | | | | | |
| b01 | 61 | 5 | 57 | 57 | 1.00 | 59 | 1.04 |
| b02 | 45 | 4 | 27 | 25 | 0.93 | 32 | 1.19 |
| b03 | 351 | 30 | 272 | 213 | 0.78 | 263 | 0.97 |
| b04 | 710 | 66 | 588 | 458 | 0.78 | 569 | 0.97 |
| b05 | 763 | 34 | 702 | 695 | 0.99 | 669 | 0.95 |
| b06 | 71 | 8 | 46 | 34 | 0.74 | 52 | 1.13 |
| b07 | 650 | 49 | 545 | 471 | 0.86 | 560 | 1.03 |
| b08 | 231 | 21 | 169 | 169 | 1.00 | 180 | 1.07 |
| b09 | 226 | 28 | 169 | 164 | 0.97 | 174 | 1.03 |
| b10 | 268 | 17 | 223 | 205 | 0.92 | 240 | 1.08 |
| b11 | 865 | 31 | 746 | 723 | 0.97 | 759 | 1.02 |
| b12 | 1390 | 119 | 1291 | 1279 | 0.99 | 1365 | 1.06 |
| b13 | 415 | 53 | 342 | 342 | 1.00 | 375 | 1.10 |
| [OpenCores] | | | | | | | |
| wb_builder | 438 | 18 | 420 | 420 | 1.00 | 431 | 1.03 |
| stepper | 455 | 39 | 254 | 159 | 0.63 | 187 | 0.74 |
| ss_pcm | 562 | 87 | 436 | 436 | 1.00 | 451 | 1.03 |
| usb_phy | 769 | 98 | 544 | 540 | 0.99 | 560 | 1.03 |
| sasc | 912 | 117 | 714 | 713 | 1.00 | 744 | 1.04 |
| i2c | 1257 | 128 | 956 | 956 | 1.00 | 1013 | 1.06 |
| spi | 1273 | 229 | 3013 | 3012 | 1.00 | *3106 | 1.03 |
| simple_spi | 1273 | 132 | 943 | 936 | 0.99 | 946 | 1.00 |
| systemcdes | 3227 | 190 | 3246 | 3215 | 0.99 | *3404 | 1.05 |
| tv80 | 8992 | 359 | 8346 | 8338 | 1.00 | *8693 | 1.04 |
| mem_ctrl | 9532 | 1069 | 8399 | 8327 | 0.99 | *8672 | 1.03 |
| systemcaes | 10965 | 670 | 9270 | 9250 | 1.00 | *9476 | 1.02 |
| usb_funct | 18397 | 1734 | 16041 | 15962 | 1.00 | *16560 | 1.03 |
| pci | 24252 | 3325 | 21020 | 20951 | 1.00 | *21593 | 1.03 |
| | | | | | Avg: 0.94 | Avg: 0.76 | |

Table 1: Circuit characteristics of ITC’99 and OpenCores benchmarks and literal counts after simplification using SDC cubes with 3 literals or fewer. Ratios between literal counts for first and smallest unrolled frames are also given. The last two columns show literal counts for *script.rugged* and ratios of these counts to the first unrolled frame. * indicates that SIS could not perform *full_simplify*.

computation. (The same fanin limit is used.) On several of the OpenCores circuits, the number of BDD nodes exceeded SIS’s limit, and *full_simplify* was not performed. The literal counts for these circuits are marked with a *. Note that in almost every case (23 out of 27) the literal count from our DC computation and simplification of the first frame is lower than the *script.rugged* count. This indicates that little optimization potential is lost by restricting the cubes to 3 literals.

Figure 8 illustrates the simplification of successive frames in the ITC’99 circuits and the OpenCores *stepper* circuit. (For simplicity, the other OpenCores modules, which did not simplify significantly, are omitted from the plot.) The literal count of each frame is given relative to the first frame. The plot shows that when significant reduction is achieved, few startup frames are needed. Many circuits show little simplification at all.

In Figure 9, gate count is plotted against minimum relative literal count. For small circuits (fewer than 1000 gates), the counts are not correlated, but large circuits consistently fail to simplify significantly. A possible explanation is that the large circuits are large because they include datapath elements many bits wide; datapaths such as ALUs have few or no unreachable states.

In light of the average simplification achieved in BMC by related techniques in [6], our results are surprisingly low. A few key differences help explain the disparity: First, the BMC method exploited equivalences between AIG vertices in different frames. To

