

Sequential Rewriting and Synthesis

Robert Brayton Alan Mishchenko

EECS Department, University of California, Berkeley, CA 94720
{brayton, alanmi}@eecs.berkeley.edu

Abstract

Present industrial applications require that logic synthesis should be scalable, which means that the algorithms used should have essentially linear complexity in circuit size. For this, we propose sequential rewriting and mapping with sequential choices. Sequential rewriting includes combinational rewriting and inherits its low computational complexity. During sequential rewriting, a history And-Inverter-Graph (HAIG) can be constructed, which compactly records all logic structures created, in a graph using sequential choice nodes. These combine both combinationally and sequentially equivalent nodes in a single equivalence class. The HAIG, which represents multiple structures, can be used to substantially improve the quality of technology mapping. An invariant is maintained for discarding rewriting steps for which an equivalent initial state does not exist. Experiments indicate that the HAIG can be constructed with less than 5% overhead in runtime for very large practical circuits.

1 Introduction

Some sequential synthesis operations (such as integrated retiming and resynthesis) can result in considerable reductions in delay (e.g. see [14]), but most industrial CAD tools do not automate these because of poor scalability and complex verification. However, DAG-aware combinational rewriting is scalable and effective [15]. The current paper extends these ideas to sequential circuits, maintaining scalability and effectiveness.

Contributions. The contributions of this paper are:

1. Combinational rewriting [15] is extended to the sequential case. The resulting algorithm does not inherit any of the inefficiencies and constraints of classical combinational synthesis, which is based on fixed latch boundaries.
2. Experiments show that sequential rewriting is only about 20% slower than the combinational rewriting and therefore can be repeated many times.
3. Every two-input AND node and the logic functions associated with *all* of its k -feasible cuts are considered for sequential rewriting.
4. A cut in a sequential AIG can cross the latch boundary (sequential cut), making the rewriting “sequentially transparent”.
5. Sequential AIGs are used to represent the logic network, which allows the rewriting to exploit the transparent register boundary.
6. The notion of sequential choices is introduced. It is shown how they can be merged with combinational choices and used in technology mapping. Recent developments for mapping show how this can be done in linear time [17].

Related Works. Several works of note on sequential synthesis have connections to the sequential rewriting of the present paper. All involve retiming and logic transformations that alter the circuit structure.

In [12], retiming and resynthesis (R&R) was proposed, in which the latches are moved out of the way as much as possible to allow for more combinational synthesis.

In [18], one retiming and one set of combinational logic transformations was considered (in either order). A *retiming invariant* was proposed, which conjoins functional relations among latch values derived from atomic retiming moves. The *rewriting invariant* of the present paper generalizes this.

In [8], labels on variables were introduced where a variable separated by a latch had labels differing by 1. The operations, elimination, extraction, resubstitution and decomposition were extended for logic expressions with labels.

In [4], an expression whose support is composed entirely of latch variables (retimeable expression) was introduced. Synthesis was based on finding such expressions during an extraction process.

In [1], retiming and a limited form of rewriting was done on sequential AIGs. Their focus was extreme efficiency, to be done on-the-fly while constructing the AIG. This is similar to *sequential structural hashing*, described in Section 3. In addition, *latch dragging* was used to expose large AND clusters of nodes, which were re-factored to minimize the number of included latches.

Organization. Section 2 surveys AIGs and Section 3 presents sequential AIGs and algorithms operating on them, such as structural hashing. Section 4 extends rewriting to the sequential case in a transparent way. Section 5 describes the “history AIG” (HAIG), sequential choices and their use in technology mapping. Section 6 shows how the HAIG can be used to derive an initial state of the final implementation equivalent to a given initial state of the original

design. Section 7 presents experimental results and Section 8 concludes. Most proofs are omitted, but can be obtained in a separate technical report.

2 Combinational Networks and AIGs

A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs). The output of a node may be an input to other nodes called its *fanouts*. The inputs of a node are called its *fanins*. If there is a path from node a to b , then a is in the *transitive fanin* of b and b in the *transitive fanout* of a . The transitive fanin of b , $TFI(b)$, includes node b and the nodes in its transitive fanin, including PIs. The *transitive fanout* of b , $TFO(b)$, includes node b and the nodes in its transitive fanout, including POs.

A (combinational) *And-Inverter-Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. *Structural hashing* of AIGs ensures that, for each pair of nodes, there is at most one AND node having them as fanins (up to permutation). It is performed by one hash-table lookup when AND nodes are created and added to the AIG manager. An AIG is often *balanced*, to reduce the number of AIG levels, by applying the associative transform, $a(bc) = (ab)c$. Both structural hashing and balancing are performed in one topological sweep from the PIs and have linear complexity in the number of AIG nodes.

The *size (area)* of an AIG is the number of its nodes; the *depth (delay)* is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations on an AIG is to reduce both area and delay.

An efficient software implementation of an AIG package is similar to that of an efficient BDD package [6]. Inverters are represented as flipped pointers to the AIG nodes. AIG nodes have reference counters, indicating the number of fanouts. AIG packages also support unique tables to ensure that there is only one node with the given fanins. However, AIGs are not canonical, since the AND-decomposition of a logic function is not unique; thus the unique table only guarantees structural canonicity within one logic level (i.e. structural hashing).

Definition. A *cut* C of node n is a set of nodes, called *leaves*, such that each path from a PI to n passes through at least one leaf. Node n is called the *root* of cut C . The *cut size* is the number of its leaves. A *trivial cut* is the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed K . A cut is *dominated* if another cut of the same node is contained in the given cut. The *volume* of a cut is the total

number of nodes encountered on all paths between Node n and the cut leaves.

3 Sequential AIGs

Sequential AIGs add sequential elements, which can be seen as technology-independent D-latches with one input and one output. All latches are assumed to have the same clock, which is omitted in the AIG representation.

Most sequential AIGs [1][14] represent latches and their initial values as attributes on AIG edges, similar to [11]. For reasons of computational efficiency, we represent latches in the AIG explicitly as one-input “boxes”. In what follows, “sequential AIG” refers to this version.

Structural hashing of sequential AIGs [1] is similar to that of combinational AIGs. A node is a PI, a PO, an AND-gate, or a latch. Structural hashing maintains an invariant that no two nodes have the same fanins. In addition, the following two invariants make sequential AIGs “more” canonical to increase compactness.

1. If a latch has a complemented fanin, the complemented attribute and complemented initial value are propagated to the fanout.
2. If an AND-node has two latch fanins, they and their initial values are retimed forward.

Sequential structural hashing may propagate changes to the fanouts and lead to rehashing large parts of the AIG. For example, retiming a latch forward over a node may result in a sequence of other retimings to maintain Invariant 2.

Structural hashing consists of the basic atomic steps:

- (1) merge two nodes with the same pair of input(s),
- (2) move two latches forward across an AND gate,
- (3) switch the phases of a latch input and output to make the input positive,
- (4) replace an AND node with both inputs the same by that input,
- (5) replace an AND node where one input is the complement of the other by 0,
- (6) propagate constants, and
- (7) remove a non-PO node if it has no fanout.

The increased canonicity is demonstrated by Theorems 3.1 and 3.2 below.

Theorem 3.1: The result of structurally hashing an AIG network is independent of the order in which the atomic steps are performed.

Definition: $str(A)$ is the unique (by Theorem 3.1) result of structurally hashing AIG A , where constant propagation is not performed.

Theorem 3.2 AIG B is a retimed version of AIG A if and only if $str(A)$ is graph-isomorphic to $str(B)$.

This represents an especially simple way to verify retiming done on an AIG.

Definition. A *cut* in a sequential AIG for a node n , is a set of nodes (AND nodes, latches, or PIs) in $\text{TFI}(n)$, which

- partitions $\text{TFI}(n)$ into nodes on paths from the cut to n (called the cone), and the rest,
- the cone nodes, not in the cut set, have inputs only from the cut set or other cone nodes, and
- the cone does not contain a loop.

Node n is called the root of the cut.

Definition. The *leaves* of a cut are its nodes, each indexed by the number of latches on any path from the root to the node.

For example, if node b is in a cut and there is a path from b to the root with no latches, and another path from b with one latch, then the cut leaves include both b^0 and b^1 .

Definition. The *sequential depth* of a cut is the maximum number of latches on a path from any cut node to the root.

A combinational cut has the property that the value of the root n is a unique function of the cut variables. For a sequential cut, the value of n is a unique combinational function of current and previous (up to sequential depth) values of the cut variables.

4 Sequential Rewriting

4.1 Rewriting of Sequential AIGs

DAG-aware rewriting [15] comprises a set of fast greedy algorithms to minimize AIG size for combinational circuits. One such algorithm considers each 4-input cut rooted at each node and tries to replace the current logic structure of the cut cone by pre-computed logic structures implementing the same logic function. If there is an improvement in the number of AIG nodes (considering sharing with existing nodes) and (optionally) no increase in the number of AIG levels, the current cone's logic structure is modified appropriately. This is repeated for the next node in a topological order. A detailed discussion of the algorithm is presented in [5][15].

Rewriting is modified for use in sequential circuits. Sequential rewriting for a given node involves the following *conceptual* atomic steps illustrated in Figure 4.1. For a given sequential cut of a node,

- Duplicate nodes on the reconvergent paths with different numbers of latches.
- Retime all latches backward to the cut.
- Do combinational rewriting (as done in [15]) of the resulting combinational function in terms of the cut leaf variables.
- Structurally hash the result.

Thus sequential rewriting is theoretically a subset of retiming and resynthesis. Note that if the same variable appears in more than one time frame, duplication is necessary to retime all latches to the inputs of the cut. Then, pure combinational rewriting can be done on the cone. Finally, all latches are retimed maximally forward while sharing logic (i.e. sequential structural hashing as discussed in Section 3). Some of the previous duplication is reclaimed in the hashing process.

We comment that rewriting need not be restricted to one node at a time. For example, the window method for using ODCs [13], can be extended to sequential circuits by retiming the latches to the inputs of the window (using duplication if necessary). This is simply another type of sequential *rewriting* operation, but where the window outputs are multiple nodes being rewritten simultaneously. Thus, ODC based simplification is just another type of rewriting and includes the sequential case.

4.2 The Rewriting Invariant

After a rewriting step, the two associated root nodes have identical combinational functions of cut leaves. This does not mean that they always evaluate to the same value, since the leaf variables involve previous values of the cut variables. In general, each root node is a combinational function of the current values of those latches, r , which lie between the root and the cut, plus some of the cut variables, x . In particular,

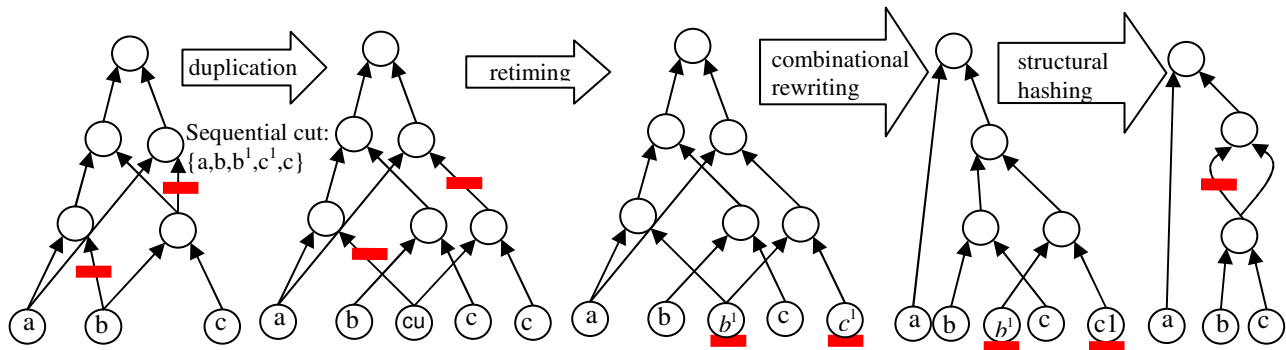


Figure 4.1. Sequential rewriting illustrated.

they are functions of the latches and cut variables reachable from the root without passing through a latch. If the two root nodes of a rewriting are denoted n and \tilde{n} , we can express them combinatorially as $n = g(r, x)$ and $\tilde{n} = \tilde{g}(r, x)$ where some of the r and x may not appear.

Generally, $g(r, x) \neq \tilde{g}(r, x)$ for arbitrary values of r and x , but we will derive a relation (called a *rewriting invariant*) among the latch values, r , which imply the equality at the root nodes. We use Example 4.1 to illustrate this. Later, in Section 6, we use this invariant to filter out some of the rewriting steps so that an equivalent initial state for the synthesized machine can be computed from a given initial state of the original machine.

Example 4.1: In Figure 4.2, the function at the nodes n and \tilde{n} in terms of the leaf variables is

$$n = (a^2 + b^2)(b^1 + c)(b^2 + c^1) = \tilde{n} = (a^2 c^1 + b^2)(b^1 + c).$$

The latches involved are denoted $\{r, s, t, u, x, y\}$. We can express each latch output in terms of the cut variables at different times, obtaining a parametrized relation among the latches,

$$\begin{aligned} (s = a^1 + b^1)(u = b^1)(y = a^1) \wedge \\ (r = a^2 + b^2)(t = b^2 + c^1)(x = a^2 c^1 + b^2). \end{aligned}$$

Examining these we see a relation among the latch values $x = rt$ and $s = y + u$. We can obtain this relation by existential quantification of non-latch variables:

$$\begin{aligned} \exists_{a^1 b^1 a^2 b^2 c^1} (s = a^1 + b^1)(u = b^1)(y = a^1) \wedge \\ (r = a^2 + b^2)(t = b^2 + c^1)(x = a^2 c^1 + b^2) \\ = (s = y + u)(x = rt), \end{aligned}$$

which is called the rewriting invariant of this step.

In general, the *rewriting invariant* is derived as follows.

1. Express each latch in terms of the cut variables in different time frames. If π_w denotes the cut variables in different timeframes, then each latch can be expressed as $r_i^w = f_i(\pi_w)$.
2. The rewriting invariant for rewriting step w is $I_w(r^w) = \exists_{\pi_w} \prod_i [r_i^w = f_i(\pi_w)]$.

Theorem 4.1: Let n_w and \tilde{n}_w denote the two root nodes of rewriting step w . Then $I_w \Rightarrow (n_w = \tilde{n}_w)$, i.e. if I_w holds, then the two node functions are combinatorially the same.

Theorem 4.2: If I_w holds in one clock cycle, then it holds in the next clock cycle.

Proof: Let x^0 denote the vector of cut variables at the current time, x^1 those at one time frame earlier, etc. If k is the sequential depth of step w , then the two

root nodes and all signals in the cones can be expressed as combinational functions of x^0, x^1, \dots, x^k . Let r be the outputs of the latches involved in step w , and s be the corresponding inputs. Writing s_i as a function of the variables in the current time frame, we have $s_i = g_i(r^0, x^0)$. Thus, $r_i = g_i^1(r^0, x^0) = g_i(r^1, x^1)$. Now form the following two expressions:

$$\begin{aligned} J &= \prod (r_i = g_i(r^1, x^1)) \\ \hat{J} &= \prod (s_i = g_i(r^0, x^0)) \end{aligned}$$

In each of the g_i in J , recursively expand the latches r_j until only x^1, \dots, x^k remain, obtaining

$$J = \prod (r_i = f_i(x^1, \dots, x^k)).$$

Note that $I_w(r) = \exists_{x^1, \dots, x^k} J$. Now suppose values, ρ , in the latches r satisfy $I_w(\rho) = 1$. Then there exists $\tilde{x}^1, \dots, \tilde{x}^k$ such that $\rho_i = f_i(\tilde{x}^1, \dots, \tilde{x}^k)$. The latch inputs values are $\{\sigma_i = g_i(\rho, x^0)\}$. Substituting these for r^0 into \hat{J} we obtain,

$$\begin{aligned} \hat{J} &= \prod (\sigma_i = g_i(\dots f_j(\tilde{x}^1, \dots, \tilde{x}^{k-1}), \dots, x^0)) \\ &= \prod (\sigma_i = f_i(x^0, \tilde{x}^1, \dots, \tilde{x}^{k-1})) \end{aligned}$$

Note that this has the same functional form as J if we associate $x^0 \sim x^1, \tilde{x}^1 \sim x^2, \dots, \tilde{x}^{k-1} \sim x^k$. Defining $\hat{I}_w(s) = \exists_{x^1, \dots, x^k} \hat{J}$, we see that $\hat{I}_w(\sigma) = 1$ because there exists x^1, \dots, x^k , namely $x^0, \tilde{x}^1, \dots, \tilde{x}^{k-1}$. Since σ will be the values in r on the next cycle, we conclude that $I_w(r)$ is a simple inductive invariant. **QED**

Example 4.1 (continued): To show that $n = \tilde{n}$, when I_w holds, we see that I_w implies that $x = rt$. Since $n = rt(u + c)$ and $\tilde{n} = x(c + u)$, then $n = \tilde{n}$. Note that in general $n \neq \tilde{n}$.

To show that I_w is an inductive invariant, let \hat{r} denote the signal at the input to latch r , \hat{s} the signal at the input of latch s , etc. The invariant I_w rewritten in terms of these signals is denoted \hat{I}_w , i.e. $\hat{I}_w = (\hat{s} = \hat{y} + \hat{u})(\hat{x} = \hat{r}\hat{t})$. Since $\hat{r} = s$, $\hat{s} = a + b$, $\hat{t} = u + c$, $\hat{u} = b$, $\hat{x} = u + yc$, $\hat{y} = a$ then $\hat{s} = a + b = \hat{y} + \hat{u}$. Since $I_w \Rightarrow (s = y + u)$, then $\hat{x} = u + yc = (y + u)(u + c) = s(u + c) = \hat{r}\hat{t}$. Thus $I_w \Rightarrow \hat{I}_w$ and therefore in the next clock cycle, the relation I_w will hold among the latches involved.

5 Choice Nodes, Mapping and HAIG

A key part of synthesis is the technology mapping step. In this section, we show how to extend the notion

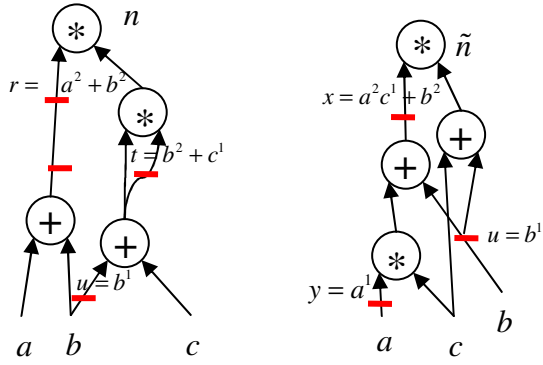


Figure 4.2. Node n (left) is rewritten, using the cut $\{a, b, c\}$, into \tilde{n} (right). $\{a^2, a^1, a, b^2, b^1, b, c^1, c\}$ are the cut variables in different time frames. Latch output signals are shown in terms of these independent variables.

of choices to sequential choices and discuss how these can be used in mapping.

5.1 Sequential Choices

To create sequential choices, each rewriting step is executed on the current AIG (called *working AIG*). The result is duplicated and added to a *history AIG* (HAIG). The original node and its rewritten version are grouped together using a *choice node*. Previously, choice nodes referred to combinational equivalent nodes. In this section, we extend the notion of a choice node to sequential equivalence.

Definition: Two nodes x and y are SE ($x \approx^{\text{SE}} y$) if they have identical combinational functions of a common sequential cut leaf variables.

Note that this definition also works combinational; hence after any rewriting, combinational or sequential, the two root nodes are SE.

Theorem 5.1: \approx^{SE} is an equivalence relation.

Theorem 5.1 allows us to rewrite a node multiple times and put all the results into a single choice node without the need to prove pair-wise equivalences. Further, all choice nodes, combinational and sequential, can be treated uniformly, i.e. there is only one type of choice node.

The HAIG can be used for technology mapping and integrated retiming (discussed in Section 5.2) and, in that context, forms a lossless synthesis database where no structure ever seen during the synthesis process is lost. The HAIG also provides a basis for scalable sequential verification (see comments in Section 8).

It should be noted that because of the choice nodes, a node in a HAIG can have many transitive fanin cones and a cut is defined as a sequential cut for any one of these cones. At a choice node, its set of cuts is the *union* of the sets of cuts of the nodes in the choice class [7]. A cut still has the property that a local

function of a node n is a function of the sequential cut leaf variables.

The process of recording rewriting steps in the HAIG is illustrated in Figure 5.1. Note that after the structure created by rewriting is copied into the HAIG, structural hashing propagates latches forward and removes duplicate logic. Thus, in Figure 5.1b, the node with fanins b and c has only one copy when the rewriting result is added to the HAIG. The blank nodes existed before rewriting and the shaded ones are the new ones not structurally similar to any pre-existing node in the HAIG.

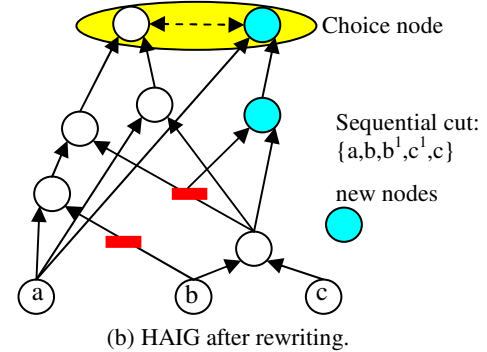
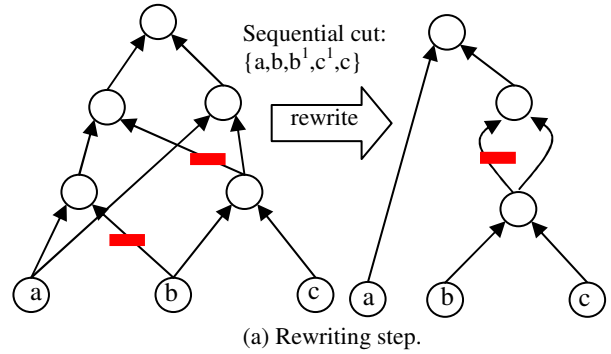


Figure 5.1. The HAIG accumulates choices.

5.2 Technology Mapping using the HAIG

Technology mapping is applied to the HAIG and is performed using sequential cuts, computed using exhaustive cut enumeration [16]. During mapping, latches are essentially ignored; the HAIG is treated as a cyclic combinational graph and sequential cuts are computed by iteration until convergence [14]. If the Boolean function of the leaf variables of a cut matches a combinational library element, the nodes between the cut and the root can be mapped using that element, even if there are latches in the cut cone. If a path from the root node to the cut leaves of a match includes a latch, the same transformation is applied during mapping as was done during sequential rewriting; we duplicate some nodes in the cone so that the latches

can be moved to the front of the cut. With the latches out of the way, the library element corresponding to the match replaces a combinational AIG subgraph (illustrated in Figure 5.2). The initial value after retiming is computed by separating retiming into two parts: (a) backward retiming of registers from under the cut and (b) retiming of registers over the mapped cuts. Computing initial value after both retimings can be combined and solved as a SAT problem.

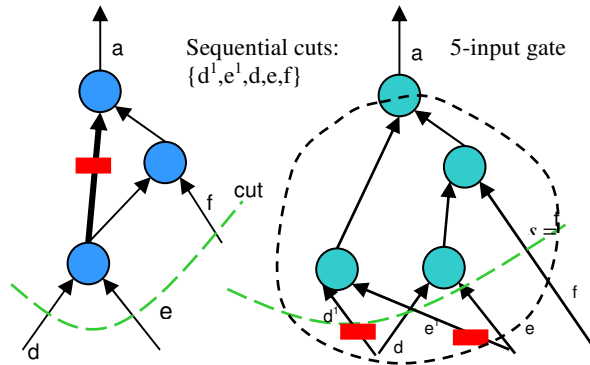


Figure 5.2. Mapping forces some duplication

A possible concern about this procedure is that during this mapping, a selection of a choice at one choice node might restrict the selection of the choice at another choice node. However, a key observation is that any choice of a sequential cut used during the mapping process forces the appropriate duplication and movement of nodes and latches to allow the mapping.

6 Initial State Computation

The final part of sequential synthesis is that of computing an equivalent initial state. Two types of initialization commonly used are:

- an initial state q is given for the initial machine D , and the synthesized machine C should start in an equivalent state \tilde{q} , i.e. $q \equiv \tilde{q}$, or
- an *initialization sequence* is given, which brings the circuit, starting at an arbitrary state (power-up state) into a known “initial” state.

Cogent arguments on why Type a) is relevant for industrial practice are presented in [2], and in this paper we only discuss this.

In general, if synthesis involves retiming, it may be impossible to derive an initial state \tilde{q} for C to make it sequentially equivalent to a given initial state q of D . Unless some retiming moves (some forward moves) are disallowed, it is known that only after cycling the machines for a certain number of clock cycles can one expect to find equivalent initial states [20].

In order to derive \tilde{q} , we can show that the rewriting steps that must be disallowed are precisely those that

do not satisfy an *accumulated rewriting invariant*. The invariant initially is the function stating that the latches, r_0 , in D have state q i.e. $r_0 = q$.

Denote $I^{w-1} \equiv \prod_{t < w} I_t$ and $P_w = (r_0 = q)I^w$, where r_0 are the latches of D . For a rewriting step, the rewriting invariant, I_w , is a relation among the latches r_w lying between the cut and the two root nodes. Partition r_w into (r_w^o, r_w^n) , where r_w^o are those that existed in the old working AIG, and r_w^n are the newly created latches. The r_w^o are part of the set of all latches R_{w-1} that existed up to the time of step w . Any initial condition on R_{w-1} , ρ_{w-1} , must satisfy $P_{w-1}(\rho_{w-1}) = 1$, which has been derived from previous rewritings. We are concerned with the existence of values ρ_w^n for the new latches r_w^n such that $P_w(\rho_{w-1}, \rho_w^n) = 1$. Otherwise this rewriting step will not be allowed since there would not exist a set of initial values for latches which can satisfy the invariant. At the last rewriting step ω , we choose one assignment (q, σ, \tilde{q}) , where \tilde{q} corresponds to the latches of C , such that $P_\omega(q, \sigma, \tilde{q}) = 1$. Note, by Theorem 4.2, if the initial condition satisfies $P_\omega(q, \sigma, \tilde{q}) = 1$, then P_ω will hold forever.

Theorem 6.1: Machine D in state q is equivalent to machine C in state \tilde{q} , i.e. $q \equiv \tilde{q}$.

Thus, the initial state q for the original design and the initial state \tilde{q} for the final implementation, are sequentially equivalent. This fact depends on the existence of a compatible set of initial states, (q, σ, \tilde{q}) , which satisfy the invariant I when the original design is given the initial state q .

Making sure that initial state values exist that can satisfy the invariant would seem to require proving that $P_w(R_w)$ is satisfiable at each step, w . However, an easier approach is to keep a current satisfying assignment, ρ , of P_{w-1} and simply try to extend this to satisfy P_w . If this is not possible, then we try to compute a new satisfying assignment of P_w and if unsuccessful, Step w is discarded. Our experiments show that little is lost in terms of optimization quality by disallowing those rewriting steps, which makes the accumulated invariant unsatisfiable. In addition, it can be shown that if q is in the cyclic core of the design, then no rewriting step has to be excluded.

7 Experimental Results

Some of presented algorithms have been implemented in the public-domain logic synthesis and verification system, ABC, Release 61118 [3].

Experiments were performed using two sets of benchmarks: (a) 8 large industrial circuits selected at random from [9] and (b) 8 large sequential miters derived from [10] for use in equivalence checking, as described in [16]. The experiments were designed to test the speed of rewriting in the presence of both sequential rewriting as well as when recording the HAIG. We were also interested in the relative growth of the HAIG size when extensive rewriting is done.

The experiments are summarized in Table 1. The first section lists benchmark statistics: the number of PIs, POs, and latches. The following two sections compare combinational and sequential rewriting in terms of AIG size and runtime (in seconds) measured on an IBM ThinkPad laptop with a 1.6GHz Intel CPU with 2Gb of RAM. The runtimes include only the specified transformations and does not include reading of the input file and constructing the HAIG.

The columns in Table 1 denoted *st*, *sts*, *rw* and *rws* are combinational and sequential structural hashing, and combinational and sequential/combinational rewriting, respectively.

The results indicate that sequential rewriting leads to an additional reduction in the AIG size, compared to combinational rewriting only. This reduction was larger for sequential miters compared to circuits used in hardware designs. We applied the two types of rewriting iteratively and observed that the script based on sequential rewriting produced on average 2-3% smaller AIGs. Sequential rewriting was fast, with only a 20% speed degradation over the combinational case. This overhead is due to the computation of additional sequential cuts and on-the-fly forward retiming performed while evaluating the gain after each rewriting step.

A separate experiment (not reported in Table 1) was conducted to measure the growth of the HAIG size after several rewriting iterations. Rewriting was performed with zero-cost replacements and *all* intermediate AIG structures were recorded in the HAIG. The HAIG size was compared with the size of the starting AIG immediately after sequential structural hashing. For the given set of 16 benchmarks, 1, 2, 4, and 8 rewriting iterations (over the entire circuit) led to a HAIG that was larger by 1.88x, 2.20x, 2.47x, and 2.71x, respectively. It was also found that constructing a HAIG increases the runtime of sequential rewriting by less than 5% on average.

These results show that:

- (a) Sequential rewriting is very fast and improves AIG size, compared to the combinational rewriting.
- (b) "HAIGing" is affordable in terms of run time and memory. The HAIG compactly represents a large

number of logic structures derived during sequential logic synthesis.

- (c) Although the reduction in *working* AIG size is small, the HAIG represent different structures at the register boundaries, which should enable better matches in this region when technology mapping is done.

These results support our on-going work in synthesis and verification, which leverages information from synthesis supplied by the HAIG to perform sequential verification.

8 Conclusions and Future Work

Combinational DAG-aware rewriting was extended to sequential circuits in a sequentially transparent way, so that the efficiency of the former is preserved. The concept of sequential choices was developed, and it was shown how a history AIG (with choices) can be constructed and used in technology mapping. Linear scalability is achieved, which allows sequential synthesis to be applied to large industrial designs.

The present work was co-developed with an equally efficient method for sequential equivalence checking. Roughly, this is based on the accumulated rewriting invariant (discussed in Section 6) being an inductive invariant of the HAIG, which contains the initial and final designs. The construction of the initial state of the final design, as given in Section 6, is such that the initial state of the HAIG satisfies the invariant and hence the invariant holds forever. These facts can be used by verification to make sequential equivalence checking have the same level of complexity as combinational equivalence checking.

All the algorithms, for both sequential synthesis and verification, are being implemented in the system ABC and extensive experiments on industrial benchmarks will be conducted. Source code will be made available.

Acknowledgements

This research was supported in part by SRC contracts 1361.001 and 1444.001, and by the California Micro program with industrial sponsors, Altera, Intel, Magma, and Synplicity.

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 60306*. <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [4] S. Bomm, N. O'Neill, and M. Ciesielski. "Retiming-based factorization for sequential logic optimization", *ACM TODAES*, vol. 5(3), July 2000, pp. 373-398.

- [5] P. Bjesse and A. Boraly, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.
- [6] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package", *DAC '90*, pp. 40-45.
- [7] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *ICCAD '05*.
http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf
- [8] G. De Micheli, "Synchronous logic synthesis: Algorithms for cycle-time minimization", *IEEE Trans. CAD*, vol. 10(1), Jan. 1991, pp. 63-73.
- [9] M. Hutton and J. Pistorius, *Altera QUIP benchmarks*.
<http://www.altera.com/education/univ/research/unv-quip.html>
- [10] *IWLS '05 Benchmarks*.
<http://iwls.org/iwls2005/benchmarks.html>
- [11] C. E. Leiserson and J. B. Saxe. "Retiming synchronous circuitry", *Algorithmica*, 1991, vol. 6, pp. 5-35.
- [12] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques", *IEEE Trans. CAD*, vol. 10, pp. 74-84, Jan. 1991.
- [13] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *DATE '05*, pp. 418-423.
- [14] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *ERL Technical Report*, EECs Dept., UC Berkeley, April 2006. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tech06_int.pdf
- [15] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *DAC '06*, pp. 532-536.
http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf
- [16] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *ICCAD '06*, pp. 836-843.
http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_ccc.pdf
- [17] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Mapping with Priority Cuts", *IWLS*, June 2007.
- [18] M. Mneimneh and K. Sakallah, "REVERSE: Efficient sequential verification for retiming", *IWLS '03*, pp. 133-139.
- [19] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *FPGA '98*, pp. 35-42.
- [20] N. Shenoy, K. Singh, R. Brayton, and A. Sangiovanni-Vincentelli, "On the temporal equivalence of sequential circuits", *Proc. DAC '92*.

Table 1. Performance of sequential AIG rewriting.

Benchmark	Statistics			AIG size				Runtime, sec			
	PI	PO	Latch	st	sts	rw	rws	st	sts	rw	rws
fip_risc8	30	83	1140	9972	9972	8128	8108	0.01	0.04	0.59	0.62
oc_aquarius	464	3328	1477	25058	25056	21346	21194	0.04	0.23	1.47	1.42
oc_cord_r2p	34	40	1015	15773	15773	11103	11103	0.02	0.07	0.70	0.84
oc_fpu	262	280	659	24932	24916	17301	17149	0.03	0.24	1.04	1.24
oc_mips	54	291	1256	20636	20636	16265	16232	0.03	0.10	1.30	1.15
oc_pavr	35	153	1231	19577	19577	15628	15378	0.02	0.11	0.88	1.04
oc_vid_det	1903	3528	3549	46433	46432	34954	34954	0.09	0.30	1.83	2.60
oc_vid_jpeg	1720	3450	3972	56601	56601	46430	46048	0.10	0.31	2.52	3.19
ac97_ctrl	84	1	5551	26701	26681	24814	21616	0.05	0.23	0.60	1.21
aes_core	259	1	1280	46304	46269	41557	41124	0.08	0.30	1.70	2.05
des_area	240	1	212	10031	10021	9152	9041	0.01	0.01	0.37	0.43
des_perf	234	1	19358	169163	164255	137413	137413	0.36	1.37	6.04	10.66
systemcaes	260	1	1880	24193	24166	21861	20730	0.03	0.10	0.55	1.18
usb_funct	128	1	4443	32977	32809	29295	28613	0.06	0.20	0.74	1.38
vga_lcd	89	1	37801	222219	222197	187422	183877	0.59	1.73	5.45	10.15
wb_conmax	1130	1	5160	97104	97017	88700	87447	0.20	0.49	1.99	3.15
Ratio				1.00	0.99	0.83	0.81	1.00	3.85	27.2	34.4