Abstract

To handle realistic circuits found in present-day industrial applications, logic synthesis should be scalable, which essentially means that the algorithms should have linear complexity in circuit size. We propose sequential rewriting, an extension of combinational rewriting to sequential circuits that inherits the low computational complexity of combinational rewriting. During synthesis, a history And-Inverter-Graph (HAIG) is constructed, which compactly records all logic structures created. The multiple structures can be used to substantially improve the quality of technology mapping and simplify otherwise intractable sequential verification. An invariant is maintained for filtering out rewriting steps, for which an equivalent initial state does not exist. Experiments performed indicate that the HAIG can be constructed in a scalable manner for very large practical circuits.

1 Introduction

Sequential synthesis can result in considerable reductions in delay (e.g. see [14]); however, most present-day industrial CAD tools do not automate sequential synthesis because of its poor scalability. On the other hand, DAG-aware combinational rewriting has been shown to be scalable and effective [15]. This paper extends these ideas to sequential circuits inheriting their scalability and effectiveness.

Several works of note in the literature on sequential synthesis have close connections to the method of sequential rewriting proposed in this paper. All these involve retiming connected with logic transformations that alter the circuit structure. In [12], retiming and resynthesis (R&R) was proposed in which retiming moves the latches out of the way as much as possible. Then the usual logic transformations such as extraction, resubstitution and node simplification, can be done on the newly exposed combinational portion of the circuit. To expose as much logic as possible, negative latches were allowed at the PIs and POs. This retiming was repeated to expose different portions of the logic. One constraint was that a final retiming must exist, which removes all negative latches.

In [8], labels on variables were introduced where two variables separated by a latch had labels differing by 1. The operations, elimination, extraction, resubstitution and decomposition were extended for logic expressions with labels. Expressions with all labels 0 correspond to combinational expressions. Synchronous division (algebraic or Boolean) was defined for dividing expression \( J \) into expression \( I \), each of which can have variables with non-zero labels. The same variable but with different labels was treated as a set of different independent variables. Then normal combinational algebraic division operations can be done. The procedure is applied iteratively, increasing all labels of \( J \) by 1 each time and then attempting algebraic division of the result until a label in \( J \) exceeds a label in \( I \) for the same variable. The notation \( J^{r} \) denoted expression \( J \), where all its variables had its labels increased by \( r \). If \( J^{(r)} \) was found to divide \( I \), then \( J \) can be rewritten as \( I = J^{(r)} + Q + R \) and a new node \( J \) can be formed with \( r \) latches at its output to form a new variable \( J^{r} \) and \( I \) is rewritten as \( I = J^{r} + Q + R \).

In [4], the concept of a retimeable expression was introduced. This is one whose support is composed entirely of latch variables. Hence, if the expression is implemented as a new node, all its inputs are latches and forward retiming can be done over the new node. After retiming, its fanins can be collapsed into the node and the result can be simplified. The proposal was to find retimable expressions during an extraction process and to evaluate the gain of each, choosing the best for implementation.

In [1], retiming and a limited form of rewriting is done on sequential AIGs as in the present paper. The focus of that work was extreme efficiency, so that it can be done on-the-fly while constructing the AIG. Retiming is done by “dragging” latches as far forward as possible, thereby exposing more of the circuit to structural hashing. This is very similar to sequential structural hashing as described in Section 3 of the present paper. In addition, the latch dragging can expose large AND clusters of nodes, which can be treated as a single multi-input AND node and re-factored using commutativity to minimize the number of included latches. The combination of latch dragging and AND clustering (combined with latch minimization in fanout stems) can reduce the number of latches and at the same time identify sequentially redundant sub-circuits.

The sequential rewriting proposed in the present paper differs in several ways from the previous work. First, combinational rewriting is extended to the sequential case in a transparent way. This transparency makes it just as efficient as combinational rewriting [15], which does not inherit any of the inefficiencies and constraints of classical combinational synthesis (such as fixed latch boundary). The second difference comes from the flexibility of using the sequential AIG structure (also recognized in [1]) to represent the logic network. In this, every two-input AND node is a node under consideration, and the logic functions considered for that node are all those obtained for any \( k \)-cut of the node. Hence, the rewriting is done on each combination of an AIG node and its multiple cuts. In contrast, for a corresponding Boolean network, a single node corresponds to one particular clustering of nodes in its transitive fanin (TFI) with the corresponding cut being the fanins of the cluster; thus only a subset of nodes and cuts are represented and are subject to logic synthesis. With sequential AIGs, a cut for a node can be a sequential one, crossing the latch boundary and making the rewriting “sequentially transparent”. Since DAG-aware rewriting is extremely fast, it can be repeated many times, resulting in networks equal or superior to those that can be obtained by classical combinational synthesis [15]. With little overhead, this capability is extended to the sequential case.

Organization. Section 2 surveys Boolean networks and AIGs. Section 3 presents sequential AIGs and algorithms operating on them, such as structural hashing. Section 4 extends DAG-aware rewriting to the sequential case in a transparent way with little cost in computational complexity. Section 5 describes the “history AIG” (HAIG), sequential choices and their use in technology mapping. In addition, we prove that rewriting creates sequentially equivalent nodes,
which can be grouped together with combinatorially equivalent nodes using choice nodes. Section 6 details how the HAIG can be used to derive a new equivalent initial state. Section 7 gives some experimental results and Section 8 concludes.

2 Combinational Boolean Networks and AIGs

Definition. 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).

Definition. 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.

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. The AIG nodes have reference counters, which indicate the number of fanouts. Both BDD and AIG packages support unique tables to ensure that there is only one node with the given fanins. Although BDDs are canonical for a given variable order, the AND-decomposition of a logic function is not unique, so the unique table guarantees structural canonicity within one logic level (i.e. structural hashing).

Definition. A cut \(C\) of node \(n\) is a set of nodes of the network, 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 of a node is the cut composed of 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 there is another cut of the same node, which is contained, set-theoretically, 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.

Definition. A local function of an AIG node \(n\), denoted \(f_{\text{local}}(x)\), is a Boolean function of the logic cone rooted in \(n\) and expressed in terms of the leaves, \(x\), which form a cut of \(n\). The global function of an AIG node is its function in terms of the PIs of the network.

Definition. Exhaustive simulation of a cone with \(k\) leaves is (bitwise) simulation with all \(2^k\) different input patterns. This produces the truth table of a local function of the leaf variables. In practice, \(k\) does not exceed 16.

3 Sequential AIGs

Sequential AIGs add sequential elements to the logic structure of combinational AIGs. The sequential elements 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.

Previous work on sequential AIGs [1]-[14], represented latches and their initial values as attributes on AIG edges, similar to [11]. For subtle reasons of computational efficiency, we choose to 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 the invariant that no two nodes have the same fanins. In addition, the following two invariants make sequential AIGs “more” canonical. The increase in the degree of canonicity is relatively small but scalable and efficient, and results in more compact AIGs.

1. A latch cannot have a complemented fanin. If one is created, the complemented attribute is propagated to the fanout and the initial value of the latch is complemented.
2. An AND-node cannot have two latch fanins. If one is created during the AIG construction, the fanin latch pairs are retimed forward over the AND-node.

Similar to the combinational case, 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 in order to maintain Invariant 2 above. Structural hashing consists of seven basic atomic steps:

1. merge two nodes when they have 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 the same inputs by that input,
5. replace an AND node with complemented inputs by 0,
6. propagate constants, and
7. remove a 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 steps are performed.

Proof: (omitted for space).

Definition. str\((A)\) is the unique (by Theorem 3.1) result of non-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\((B)\) (str\((A)\) ≅ str\((B)\)).

Proof: (omitted for space)

Definition. A cut in a sequential AIG for a node \(n\), is a set of AND nodes, latches, or PIs in TFI\((n)\), which separates \(n\) and several levels of its fanins from the rest of its transitive fanin network, i.e. the cone between the cut and \(n\) has no other inputs. This cone cannot contain a loop. Another property is that given values for the cut variables and values for the latches within the cone, the value of \(n\) is uniquely determined.

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

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 all 4-input cuts rooted at each node. For each cut, it tries to
replace the current logic structure of the cut by one of a set of 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 logic structure is modified appropriately, and the computation moves to the next node in a topological order. A detailed discussion of the algorithm is presented in [5][15].

This algorithm is modified for use in sequential circuits. Sequential rewriting for a given node involves the following conceptual atomic steps illustrated in Figure 4.1:

1. Compute a sequential cut for a node.
2. Duplicate nodes on the reconvergent paths with different numbers of latches and assign these nodes different indices.
3. Retime all latches backward to the cut leaves.
4. Do combinational rewriting on the resulting combinational function in terms of the cut leaf variables.
5. Structurally hash the result.

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 as far forward as possible in order to “canonize” the AIG to share more logic (effectively these two steps constitute sequential structural hashing as discussed in Section 3). Some of the previous duplication is reclaimed in the process.

We observe that rewriting need not be restricted to one node at a time. For example, the window method for using ODCs, originally proposed for combinational circuits [13], can be extended to sequential circuits by retiming the latches to the inputs of the window, using duplication if necessary. This can be viewed as simply another type of rewriting operation, but where the window outputs are the nodes being rewritten simultaneously. Although this is done as one operation, each output can be rewritten separately if desired; the only difference between one output at a time and all at once, might be in the effectiveness of the rewriting done, since the latter is a correlated rewriting and would share more logic. However, from a theoretical standpoint, ODC based simplification is just another type of rewriting and as such can also be extended to the sequential case.

4.2 A Rewriting Invariant

After a rewriting step, the two associated root nodes have identical combinational logic functions of a common sequential cut. This does not mean that they always evaluate to the same value as in the combinational case. In general, the root node values are combinational functions of some of the values of the latches, r, which sit between the two roots and the cut, plus some of the cut variables, x. In particular, they are functions of the latches and cut variables reachable from the roots without passing through a latch. If the two root nodes are denoted n and \(\tilde{n}\), we can express their values combinationally 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 concept. 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

\[n = (a^2 + b^2)(c^2 + b^2 + c^2) = \tilde{n} = (a^2c + b^2)(b^2 + c^2)\]

The latches involved are denoted \(\{r, s, t, u, x, y, z\}\). Retiming imposes relations among them. This is derived by propagating (using forward retiming) symbolic values from the sequential cut, obtaining the relation,

\[(s = \beta + \delta)(u = \delta)(y = \beta)(z = \delta) \land (r = \alpha + \gamma)(t = \gamma + \zeta)(x = \alpha \zeta + \gamma)\]

among the latches. Quantifying the symbolic parameters yields,

\[\exists_{\alpha}(s \neq \beta + \delta)(u = \delta)(y = \beta)(z = \delta) \land \exists_{\alpha}(r = \alpha + \gamma)(t = \gamma + \zeta)(x = \alpha \zeta + \gamma)
\]

which is the rewriting invariant of this rewriting.

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 concept. 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

\[n = (a^2 + b^2)(c^2 + b^2 + c^2) = \tilde{n} = (a^2c + b^2)(b^2 + c^2)\]

The latches involved are denoted \(\{r, s, t, u, x, y, z\}\). Retiming imposes relations among them. This is derived by propagating (using forward retiming) symbolic values from the sequential cut, obtaining the relation,

\[(s = \beta + \delta)(u = \delta)(y = \beta)(z = \delta) \land (r = \alpha + \gamma)(t = \gamma + \zeta)(x = \alpha \zeta + \gamma)\]

among the latches. Quantifying the symbolic parameters yields,

\[\exists_{\alpha}(s \neq \beta + \delta)(u = \delta)(y = \beta)(z = \delta) \land \exists_{\alpha}(r = \alpha + \gamma)(t = \gamma + \zeta)(x = \alpha \zeta + \gamma)
\]

which is the rewriting invariant of this rewriting.

In general, the rewriting invariant is derived as follows.

1. For a retiming step, conceptually retime all latches to its sequential cut.
2. Assign a symbolic parameter to each resulting latch to represent its value. Let \(\pi_n\) denote these parameters.
3. Retime all such latches forward and derive the resulting latch values symbolically in terms of \( \pi_a \), e.g. if \( r'' \) is a forward retimed latch of this rewriting step, its value is given by 
\[
r'' = f_r(\pi_a).
\]
4. The rewriting invariant for rewriting step \( w \) is 
\[
I_w(r'') = \exists_\gamma \prod [f_r(\pi_a)].
\]

**Theorem 4.1:** Let \( n_r \) and \( \bar{n}_r \) denote the two roots nodes of rewriting step \( w \). Then \( I_w \Rightarrow (n_r = \bar{n}_r) \).

**Proof:** (omitted for space)

**Example 4.1 (continued):** To show that \( n_r = \bar{n}_r \) when \( I_w \) holds, we see that \( I_w \) implies that 
\[
\exists_{a,b,c}(x = \alpha \beta \gamma \delta \zeta) (y = \beta \delta \xi) (z = \delta \xi) (r = \alpha \beta \gamma \delta \xi) (s = \beta \delta \xi) (t = \gamma \delta \xi).
\]
The relations, \( h_r \), between the \( \pi_a \) and \( \pi_s \) parameters are
\[
\alpha^2 = \alpha \beta, \beta^2 = \gamma, \beta^3 = \delta, \zeta = \xi.
\]
The local root node functions in terms of immediate latches and leaf variables are
\[
\pi_r = r(a + c)(b + c)(a + b + c)(b + c) \quad \pi_s = x(z + c)(a \beta \gamma \delta + c)(a \beta \gamma \delta + c).
\]
Thus, we observe that \( n_r = \bar{n}_r \).

5 Choice Nodes, Mapping and HAIG

5.1 Sequential Choices

Each rewriting step executed on the current AIG (called working AIG) and duplicated in the history AIG (HAIG), where the original node and its rewritten counterpart are grouped together using a choice node. Previously, choice nodes referred to combinational equivalence nodes. In this section, we extend the notion of choice node for rewriting of sequential circuits.

**Definition:** Two nodes \( x \) and \( y \) are SE \( (x \equiv y) \) if they have identical combinational functions of a common sequential cut.

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

**Theorem 5.1:** \( \equiv \) is an equivalence relation.

**Proof:** (omitted for space)

**Theorem 5.1** allows us to rewrite a node multiple times and put all the results in a single choice node without the need to prove pair-wise equivalences. Further, all choice nodes, combinational and sequential, can be treated uniformly.

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 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 separator of 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 \( n \) can be expressed as a function of the leaves of the cut.

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 existing nodes in the HAIG.

5.2 Technology Mapping using HAIG

Technology mapping on the HAIG can be 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 cuts are computed by iteration until convergence [14]. If the Boolean function of the leaf variables of a cut matches with a combinational library element, the nodes between the cut and the root can be mapped using that element, independent of whether the cut is combinational or sequential.

If a path from the root node to the cut leaves includes a latch, the same transformation is applied during mapping as was done during sequential rewriting; the cone is duplicated if necessary and the latches are moved to the front of the cut. Unlike traditional retiming where the graph structure is preserved, such retiming is always possible using logic duplication. With the latches out of the way, the library element replaces a combinational AIG subgraph (illustrated in Figure 5.2).
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, Figure 5.3 illustrates the independence of sequential choices. The 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.

**Figure 5.3. Independence of sequential choices.**

Figure 5.3 shows two choice groups, \( \{a,a'\} \) and \( \{b,b'\} \). The nodes \( a \) and \( b \) on the left are clearly compatible because they share a common retiming. The same is true for nodes \( a' \) and \( b' \) on the right. However, one might think that choosing gate2 for the mapping (which effectively chooses \( b' \)) would force a latch to be on the edge connecting \( d \) to \( a \) (to get a latch at the output of \( b \) to match that of \( b' \)). Then one might be tempted to conclude that gate1 could not be used because the two retimings are not compatible. In fact, this would be true for the standard version of retiming where the graph structure is invariant, but in our case node duplication can make the retimings compatible. In this case, by duplicating node \( d \), the two retimings are made compatible, allowing both gates 1 and 2 to be selected in the mapping.

### 6 Initial State Computation

Two types of initialization (or their combination) are used in practice. One is that an initial state \( q \) is given for the initial machine and the synthesized machine should start in an equivalent state \( \tilde{q} \), i.e. \( q = \tilde{q} \). Cogent arguments on why this is relevant for industrial practice are given in [2]. In the second scenario, when a machine powers up, the latch states can be arbitrary. In this case, there is a sequence of PI values, an initialization sequence, which brings the circuit, starting at an arbitrary state (power-up state) into a known “initial” state. In this paper we only have space to address the first scenario.

In general, if synthesis involves retiming, it may be impossible to derive an initial state \( \tilde{q} \) for the final circuit C to make it sequentially equivalent to a given initial state \( q \) of the original design \( D \). 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 [18].

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 the function stating that the latches in the original machine have state \( q \). At each rewriting step, it is conjoined with the rewriting invariant \( I_q \) computed as shown in Section 4.2.

Denote \( I^{\text{rew}} = \prod_{i=1}^{D} I_i \) and \( P_q = (r_q = q) I^{\text{rew}} \), where \( r_q \) are the latches of \( D \). For a rewriting step, the rewriting invariant, \( I_q \), is a relation among the latches lying between the cut and the two root nodes. Partition the latches, \( r_q \), of \( I_q \) into \((r_0', r_0'')\), where \( r_0' \) are those of the old working AIG, and \( r_0'' \) are the newly created ones. The \( r_0'' \) are part of the set \( R_{n-1} \) of all latches that existed up to the time of rewriting step \( w \).

These must satisfy a relation \( P_{w-1}(R_w) \), which has been derived from previous rewritings. We are concerned with the existence of “compatible” set of values \( r_0' \) for the \( r_0'' \). If \( P_w(R_w) = 0 \), then this rewriting step is not allowed. Otherwise a set of consistent values exists and we can use one as an initial value for \( r_0 \). At the last rewriting step \( w \), we choose one assignment \( (q, \sigma, \tilde{q}) \), where \( \tilde{q} \) corresponds to the latches of \( C \), such that \( P_w(q, \sigma, \tilde{q}) = 1 \).

**Theorem 6.1:** \( q = \tilde{q} \).

**Proof:** (omitted for space)

The method would seem to require solving a SAT problem on \( P_w \) at each step. However, at each step, we would only keep a current satisfying assignment \( s \), of \( P_w \), and simply try to extend this to \( P_{w+1} \), which is just a matter of satisfying \( I_w \) where some latch values are restricted to agree with some in \( s \). If this can’t be done, only then do we compute a satisfying assignment of \( P_w \) and continue. If \( P_w \) is not satisfiable, the rewriting step \( w \) is not used. 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

The presented algorithms are implemented in a public-domain logic synthesis and verification system, ABC, Release 61118 [3]. The experiments are 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 the use in equivalence checking, as described in [16].

The experiments are summarized in Table 1. The first section of the table lists benchmark statistics: the number of PIIs, POIs, and latches. The following two sections of the table 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 runtime includes only the specified transformations and does not include reading of the input file and constructing HAIG.

The columns in Table 1 are denoted as follows: “st” and “sts” are combinational and sequential structural hashing, respectively; “rwv” and “rws” are combinational and sequential rewriting, respectively. The results indicate that sequential rewriting leads to an additional reduction in the AIG size, compared to combinational rewriting. This reduction is 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. Although it may seem that this is a minor reduction, it is important in many applications targeting small AIG size. Note that sequential rewriting is 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 HAIG size after several rewriting iterations. Rewriting was performed with zero-cost replacements while recording in HAIG all intermediate AIG structures. 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 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.

The experimental results show that sequential rewriting is very fast and improves AIG size, compared to the combinational rewriting. "HAIGing" is affordable in terms of runtime and memory. The HAIG compactly represents a large number of logic structures derived during sequential logic synthesis. These results enable our on-going work in synthesis and verification, which uses the HAIG to perform (a) integrated technology mapping and (b) sequential verification using information from synthesis.

8 Conclusions and Future Work

This paper has extended combinational DAG-aware rewriting to sequential circuits in a sequentially transparent way, so that the efficiency of the former has been preserved. The scalability achieved, allows sequential synthesis to be applied to large industrial designs. However, this is just half of the story because without equally efficient sequential verification, sequential synthesis will not be adopted by industry.

Therefore, the present work has been co-developed with an equally efficient method for sequential equivalence checking. Roughly, the theory is based on the fact that the accumulated rewriting invariant (discussed in Section 6) is an inductive invariant of the product machine of the initial and final designs. The proof of sequential equivalence can be seen as an extended resolution proof with the new variables, required for the extension, furnished by the HAIG. The construction of the initial state of the final design as given in Section 6 is such that the extended initial state of the product machine satisfies the invariant. As shown in Section 4.2, the invariant implies that the root nodes have equal functions. Since the POs of the two machines share such root nodes, the POs must agree forever if the machine is started in this initial state, proving sequential equivalence.

All the algorithms for both sequential synthesis and verification are being implemented in the system ABC and will be made available in source code. The proofs supporting our theory exist and have been studied carefully by several independent experts.

References

Table 1. Performance of sequential AIG rewriting.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Statistics</th>
<th>AIG size</th>
<th>Runtime, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PI</td>
<td>PO</td>
<td>Latch</td>
</tr>
<tr>
<td>rip rushing</td>
<td>40</td>
<td>83</td>
<td>1140</td>
</tr>
<tr>
<td>oc aquarius</td>
<td>454</td>
<td>378</td>
<td>1772</td>
</tr>
<tr>
<td>oc cord r2p</td>
<td>30</td>
<td>67</td>
<td>1015</td>
</tr>
<tr>
<td>oc fpu</td>
<td>256</td>
<td>291</td>
<td>0.65</td>
</tr>
<tr>
<td>oc mips</td>
<td>54</td>
<td>126</td>
<td>1526</td>
</tr>
<tr>
<td>oc pavr</td>
<td>35</td>
<td>153</td>
<td>1231</td>
</tr>
<tr>
<td>oc vid det</td>
<td>1903</td>
<td>3528</td>
<td>3549</td>
</tr>
<tr>
<td>oc vid jpeg</td>
<td>1720</td>
<td>3450</td>
<td>3972</td>
</tr>
<tr>
<td>ac97 ctrl</td>
<td>84</td>
<td>51</td>
<td>5551</td>
</tr>
<tr>
<td>aes core</td>
<td>259</td>
<td>1280</td>
<td>1280</td>
</tr>
<tr>
<td>des area</td>
<td>240</td>
<td>121</td>
<td>10021</td>
</tr>
<tr>
<td>des perf</td>
<td>234</td>
<td>19358</td>
<td>164255</td>
</tr>
<tr>
<td>systemecaes</td>
<td>260</td>
<td>1880</td>
<td>24193</td>
</tr>
<tr>
<td>usb funct</td>
<td>128</td>
<td>4443</td>
<td>32809</td>
</tr>
<tr>
<td>vga lcd</td>
<td>89</td>
<td>37801</td>
<td>222197</td>
</tr>
<tr>
<td>wb conmax</td>
<td>1130</td>
<td>5160</td>
<td>97104</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.00</td>
<td>0.99</td>
<td>0.83</td>
</tr>
</tbody>
</table>