Optimizing Sequential Verification by Retiming Transformations

Gianpiero Cabodi[†] Stefano Quer[†] Fabio Somenzi[‡]

[†]Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

[‡]University of Colorado
Dept. of Electrical and Computer Engineering
Boulder, CO

Abstract

Sequential verification methods based on reachability analysis are still limited by the size of the BDDs involved in computations. Extending their applicability to larger and real circuits is still a key issue.

Within this framework, we explore a new way to improve symbolic traversal performance, working on the representation of state sets. We exploit retiming to reduce the number of latches of a FSM, and to relocate them in order to obtain a simplified state set representation. We consider retiming as a temporary state space transformation to increase the efficiency of sequential verification. We discuss it as a state space transformation and we formally analyze the conditions under which such a transformation is equivalence preserving for a given property under verification.

We lower image computation cost, and we reduce the size of BDDs representing intermediate results and state sets. Experimental results show considerable memory and time improvements on some benchmark and home made circuits.

1 Introduction

State-of-the-art approaches for reachability analysis and formal verification of circuits modeled as Finite State Machines (FSMs) exploit symbolic techniques based on Binary Decision Diagrams (BDDs).

Given the transition relation of a system, $\mathsf{TR}(x,y)$[1], and a set of states, $F(x)$, the set of states reachable in one step from the states in F, $T(y)$, is computed as

$$T(y) = \textsc{Image}(\mathsf{TR}(x,y), F(x)) = \exists_x(\mathsf{TR}(x,y) \cdot F(x)) \quad (1)$$

This is the core computation of all symbolic reachability and sequential verification algorithms. But even symbolic techniques reach their limits on large practical examples. Several improvements have thus been proposed to the basic idea, in order to deal with realistic circuit sizes. Among the other, we remember partitioned forms, dynamic variable reorderings, approximate traversal/verification strategies, abstractions of sub-components, and, more recently, guided searches.

[1] In the sequel we use TR, w, x and y to indicate respectively the transition relation, primary inputs, present state variables, and next state variables.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

In this paper we explore a different way of improving traversal performance, working on the representation of state sets. We exploit *retiming* to *reduce* the number of latches of a FSM, or to simply *reposition* them, to obtain smaller BDDs for the states of the transformed FSM.

Retiming has been introduced to reposition latches across the combinational logic in a sequential circuit. It is used in logic synthesis, in order to minimize the clock period, the number of latches, or to meet a given clock period while minimizing the number of latches. By definition, retiming preserves the input-output behavior of the sequential circuit.

Our use in verification with performance enhancement purposes, represents a new application of retiming. Previous works in sequential verification have used retiming with different goals. In [1, 2, 3] retiming is exploited to infer structural similarities to reduce sequential verification to a combinational equivalence proof. By contrast, we completely work in the sequential domain as we see retiming as a temporary transformation to be applied to the FSM and to the property as well. In [4] a restricted form of retiming is used to collapse registers driven by clocks of different phases. We use general retiming, and we address the issue of property retiming. Finally, in [5] retiming is used to extract as many peripheral latches as possible from the circuit. Though this is useful, it is not the focus of our work.

2 Motivation and Overview of the Presented Approach

We propose a technique to reduce the size of BDDs encountered in sequential verification. Our approach originates from the intuition that retiming may be effective as an equivalence preserving transformation to minimize the number of latches, and/or exploring alternative ways (more efficiently in terms of BDD size) of coding the information stored in latches. Retiming is usually applied to automatically designed circuits to target optimal clock cycle, area, or low-power, rather than minimum number of latches. We partially alter the above assumptions, by considering the retiming method as a *temporary* transformation to improve the efficiency of sequential verification. We can thus disregard, for instance, cycle time and area, and concentrate on latch minimization, with the aim of compacting the amount of reachable states and the size of the BDDs representing them. Moreover, we can explore different heuristics and cost measures for given latch positions, oriented to more efficient symbolic operations, rather than better hardware implementation. We can finally partially mod-

ify the equivalence criteria of the FSM transformation, which are again bound by the correctness of a verification task, not by physical implementation: A given input signal could, for instance, be artificially delayed or anticipated with respect to its original timing, thus modifying the input output behavior, but preserving the verification task. (See Section 4.)

Example 1 Let us consider a piece of a circuit, reported in Figure 1(a), containing an ALU with two input/operand registers, A and B, and an output/result one, Out. The ALU operation is selected by the *op-code* field of an Instruction Register IR. All registers are controlled by enable inputs, and the usual sequence of operations is: Fetch the instruction, load the operand registers, execute the operation, write the result in Out. A

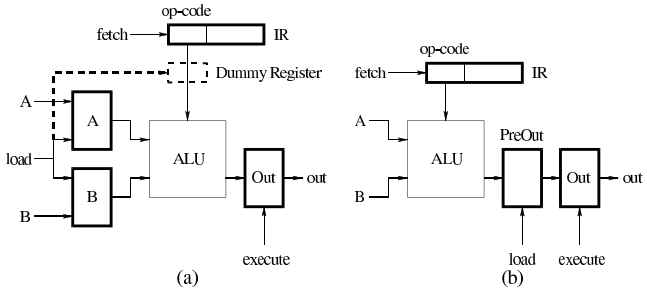

Figure 1: A Retiming Example with insertion of a "dummy" register.

relevant part of the reachable states is characterized by a tight correlation among the values of all the above registers: Out is the result of the operation specified by IR, and applied to the A and B operands. The BDDs representing such states would be reduced by retiming A and B in the forward direction across the ALU:

- A single register ($PreOut$) would replace the A and B registers, with any new value stored in $PreOut$ equivalent to sets of values previously stored in A and B.
- The values stored in $PreOut$ and Out would be equal in all operation steps, except after loading $PreOut$ (A and B in the original circuit).

Notice that a backward retiming of Out would also benefit from the second point, but not from the first one.

Figure 1(b) reports such a retimed circuit. Since retiming of latches with load enable inputs is allowed only for a set of latches sharing the load input, a dummy register (with the same enable as A and B) is inserted on the *op-code* input of the ALU. This insertion is represented with dotted lines in Figure 1(a). In Section 4.1 we show that it is permissible.

In view of the previously described goals, we implemented an *ad hoc* retiming procedure and we decided not to work on gate level net-lists, but to retime latches across entire next state functions, seen as black boxes. Although this choice reduces the available latch positions, it has some practical advantages, which are discussed in the next section.

We thus find and apply a retiming transformation by working on a FSM represented by its next state transition function. We then produce a transition relation, usually in conjunctively clustered form, to be used for the following verification tasks. The method has similarities with known published techniques on retiming, but we propose (and we rely on) some different optimality criteria. It also shares some of the inspiring ideas of works doing equivalence preserving transformations with the aim of simplifying a verification process. Concerning the particular aspect of retiming applied to sequential verification, both the theoretical framework, and the application we propose are new.

3 Retiming for Verification

We describe in this section retiming as a state space transformation oriented to simplify sequential verification operations. The theoretical framework and the formalism we propose are quite general and not limited to retiming. They could be applied with minor modifications to other FSM transformations, like, for instance, state minimization.

Retiming is exploited in logic synthesis as an equivalence preserving optimization of sequential circuits. It usually works on a net-list made of logic gates and latches, whereas in our verification framework we prefer using next state functions as basic black blocks for the retiming procedure. Although limiting the number of possible retimings, this has the following desirable consequences:

- It may be applied as a preprocessing step of formal verification, directly on the BDD representation of next state functions, without requiring any structural knowledge of the gate level net-list.
- Automatic detection of latch enables is possible by directly working on BDD representations (see Section 4.1).
- The complexity of the retiming problem is much lower than on the original net-list.

Working on gate level net-lists is an alternative option, which is not excluded from our theoretical and practical framework, provided that proper notational changes are done to all formulas, now using next state functions as basic blocks.

An FSM M is given by

$$M = (I, O, S, \delta, \lambda, S_0)$$

where I is the set of input variables, O the set of output variables, S the set of states, δ the next-state function, λ the output function, and S_0 the set of initial states. Input, present, and next state variables are denoted w, x, and y. TR is the transition relation, defined as

$$\mathsf{TR}(w, x, y) = \prod_i (y_i \equiv \delta_i(w, x))$$

Let us call M the original FSM and $\widehat{M}$ the retimed one, with $\mathsf{TR}(w, x, y)$ and $\widehat{\mathsf{TR}}(w, \widehat{x}, \widehat{y})$ their transition relations.

We can view retiming as a state re-mapping applied to the FSM. Re-mapping is expressed by the relation $\rho(x, \widehat{x})$ applied to the state spaces and returning 1 for any couple $(x, \widehat{x})$ of corresponding states, i.e., the $\widehat{x}$ state in $\widehat{M}$ is equivalent to state x in the original FSM M. In the case of retiming, state equivalence is strictly related to the nature of the transformation, which is proved to preserve the input–output behavior of a sequential circuit. In the sequel we will omit discussing conditions for maintaining the sequential behavior through the ρ transformation, which are implicit with retiming, and should be specifically addressed for more general cases.

A retiming process can be decomposed in a sequence of steps where any retimed latch is only moved across one single block

either in the forward or in the backward direction. The retiming relation of such a step can be computed through the following formula:

$$\rho(x,\widehat{x}) = \prod_{i\in N}(\widehat{x}_{N,i} = x_{N,i}) \cdot \prod_{j\in F}(\widehat{x}_{F,j} = \delta_{crossed}(x)) \cdot \prod_{k\in B}(x_{B,k} = \delta_{crossed}(\widehat{x}))$$

where N is the set of latches *not* retimed, i.e., kept in their original position, F is the set of *forward* retimed latches (in $\widehat{M}$, i.e. after retiming), and B is the set of *backward* retimed latches (in M, i.e. before retiming). $\delta_{crossed}$ is the function of the combinational block crossed by the retimed latch. The global retiming relation of a sequence of steps can be expressed as a relational composition of the step relations. A two step retiming $\rho_{1,2}$ is computed as composition of the first (ρ_1) and second (ρ_2) retiming:

$$\rho_{1,2}(x,\widehat{x}) = \exists_{x'}(\rho_1(x,x') \cdot \rho_2(x',\widehat{x}))$$

Let us represent the sequential behavior of the FSM through its transition relation. The transition relation of $\widehat{M}$ can be computed, through retiming transformations in the present and next state spaces, as:

$$\widehat{\mathsf{TR}}(w,\widehat{x},\widehat{y}) = \exists_{x,y}(\rho(x,\widehat{x}) \cdot \mathsf{TR}(w,x,y) \cdot \rho(y,\widehat{y})) \quad (2)$$

The following example shows an application of the above concepts to a simple practical case.

Example 2 Figure 2 reports an example where retiming is applied to a circuit with two latches (L_1 and L_2). f_1, f_2 and f_3 are generic combinational blocks. Figure 2(a) shows the original circuit, whose transition relation is:

$$\mathsf{TR}(w,x,y) = ((y_1 \equiv f_1(w,x_2)) \cdot (y_2 \equiv f_2(x_1)))$$

Figure 2(b) shows the result of retiming latch L_1 forward across f_2. The L_2 latch is not retimed, so the N set is L_2, and $x_{N,1}$ ($y_{N,1}$) corresponds to x_2 (y_2). The retimed state transformation relation may be expressed as[2]:

$$\rho(x,\widehat{x}) = (\widehat{x}_{N,1} \equiv x_2) \cdot (\widehat{x}_{F,1} \equiv f_2(x_1)))$$

The resulting transition relation can be computed as:

$$\begin{aligned}\widehat{\mathsf{TR}}(w,\widehat{x},\widehat{y}) &= \exists_{x,y}(\rho(x,\widehat{x}) \cdot \mathsf{TR}(w,x,y) \cdot \rho(y,\widehat{y})) \\ &= \exists_{x,y}[(\widehat{x}_{N,1} \equiv x_2) \cdot (\widehat{x}_{F,1} \equiv f_2(x_1)) \cdot \\ &\quad (y_1 \equiv f_1(w,x_2)) \cdot (y_2 \equiv f_2(x_1)) \cdot \\ &\quad (\widehat{y}_{N,1} \equiv y_2) \cdot (\widehat{y}_{F,1} \equiv f_2(y_1))] \\ &= (\widehat{y}_{F,1} \equiv (f_2 o f_1)(w,\widehat{x}_{N,1})))) \cdot (\widehat{x}_{F,1} \equiv \widehat{y}_{N,1})\end{aligned}$$

which is exactly the expression one can get directly from the circuit:

$$\widehat{\mathsf{TR}}(w,\widehat{x},\widehat{y}) = \exists_{x,y}(y_1 \equiv f_1(w,\widehat{x}_{N,1})) \cdot (x_1 \equiv y_1) \cdot (\widehat{y}_{F,1} \equiv f_2(x_1)) \cdot (\widehat{x}_{F,1} \equiv \widehat{y}_{N,1})$$

through existential quantification.

3.1 Property Preserving Retiming

Retiming is usually viewed as an input–output equivalence preserving transformation, i.e., the input–output behavior of the FSM is preserved. In our case we are interested in property verification, so we want a given property to be maintained through a retiming transformation.

Let us consider a property $P(x)$[3]. P is a propositional formula on the state variables. The property in the retimed space is defined as

$$\widehat{P}(\widehat{x}) = \exists_x(\rho(x,\widehat{x}) \cdot P(x))$$

[2]Notice that $f_2 o f_1$ expresses function composition.

[3]A more general case is a property $P(w,x)$ function of input and state variables. We avoid input dependence here for sake of simplicity.

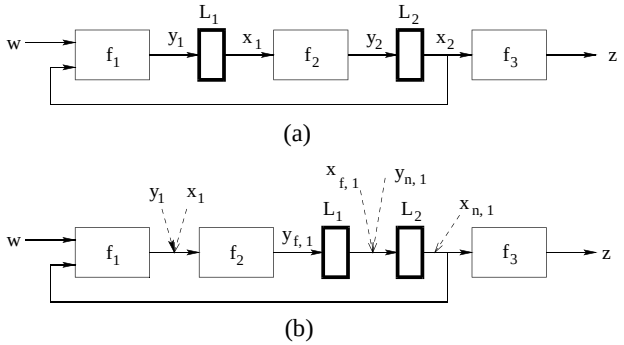

Figure 2: A retiming example.

The transformation is *property preserving* iff, given x and $\widehat{x}$ in $\rho(x,\widehat{x})$

$$P(x) \Leftrightarrow \widehat{P}(\widehat{x})$$

If all the propositional formulae appearing in a generic temporal logic formula φ are preserved, then φ is also preserved. Therefore, without loss of generality, we concentrate on properties that are propositional formulae.

Input–output equivalence is a first avenue of attack for this problem: Any property expressed as a function of primary outputs is guaranteed to be preserved by standard retiming. Whenever a property depends on internal circuit points or state elements, the property itself (in case it is a Boolean formula), or a set of Boolean sub-expressions (in case of a temporal logic property) might be regarded as output function(s) and included as such in the retiming process. Since retiming guarantees input–output equivalence, the property in the retimed FSM is equivalent to the original one.

Following the above reasoning, a wider space for retiming is available by ignoring all output functions, with the exception of the ones related to the property under verification. Moreover, input–output equivalence could be changed to a more general property preserving condition.

The following theorem expresses in a formal way a general condition relating the retiming transformation and the property under verification.

Theorem 1 Given a state space transformation defined by $\rho(x,\widehat{x})$, and a property $P(x)$, transformed as $\widehat{P}(\widehat{x}) = \exists_x(\rho(x,\widehat{x}) \cdot P(x))$, the transformation is equivalence preserving for P

$$\rho(x,\widehat{x}) \Rightarrow (P(x) \Leftrightarrow \widehat{P}(\widehat{x}))$$

if

$$P(x) = \exists_{\widehat{x}}(\rho(x,\widehat{x}) \cdot \widehat{P}(\widehat{x}))$$

Proof The direct implication follows from the definition of $\widehat{P}$. In fact, if $P(x)$ holds then $\widehat{P}(\widehat{x}) = \exists_x(\rho(x,\widehat{x}))$, which is true for all re-mapped xs (such that $\rho(x,\widehat{x})$ holds). The inverse implication is derived in a similar way from the hypothesis ($P(x) = \exists_{\widehat{x}}(\rho(x,\widehat{x}) \cdot \widehat{P}(\widehat{x}))$).

□

The theorem states a property preserving condition based on a kind of invertibility of the ρ transformation applied to the P property. The boolean condition required by the theorem should be checked for any property being verified on a retimed FSM.

Two particular cases included in the above theorem are expressed by the following lemmas. The former shows that a $1:1$

transformation is property preserving. But this is a tough constrain for retiming; hence, the latter lemma introduces a *partially* $1:1$ transformation, where the condition is required only for the latches on which the property depends.

Lemma 1 A $1:1$ transformation $\rho(x,\widehat{x})$ is equivalence preserving for any $P(x)$ property transformed as $\widehat{P}(\widehat{x}) = \exists_x(\rho(x,\widehat{x}))$.
Proof As $\rho(x,\widehat{x})$ is $1:1$ P can be expressed as an inverse transformation of $\widehat{P}(\widehat{x})$:

$$P(x) = \exists_{\widehat{x}}(\rho(x,\widehat{x}) \cdot \widehat{P}(\widehat{x}))$$

which is the hypothesis of Theorem 1.

Lemma 2 Let us define S_P ($\widehat{S}_P$) as the subspace of S described by variables in $supp(P(x))$ ($supp(\widehat{P}(\widehat{x}))$). Let us define the projection of ρ to $S_P \times \widehat{S}_P$ as

$$\rho_P(x,\widehat{x}) = \exists_{x-supp(P),\widehat{x}-supp(\widehat{P})}\rho(x,\widehat{x})$$

If ρ_P is $1:1$ the transformation is equivalence preserving.

A direct consequence of this lemma is that property $P(x)$ is preserved if we do not retime the latches it depends on.

4 Verification–Oriented Retiming

Standard retiming algorithms may be adapted with a few modifications to our verification framework. Our present solution is an *ad hoc* implementation within a BDD based traversal package.

First of all, we target minimum number of latches, without caring about clock cycle. Secondly, we also expect benefits from bringing latches face-to-face (a latch output directly feeding the input of the second one). This kind of retiming has quite often the nice effect of duplicating the information stored in two sets of latches in relevant subsets of the reachable states. See, for instance, Example 1: Forward retiming of the A and B registers has the combined effect of reducing the number of latches, and partially duplicating the information stored in the Out register. We practically favor chains of latches by reducing, through a user selectable weighting factor $w < 1$, the cost of any latch directly feeding another latch.

Further topics we describe in the following are latch enables, specific retiming rules for primary inputs, and some traversal related issues.

4.1 Latch Enables

In practical designs, different latches may be enabled under different enable conditions. Legl et al. [6] consider the problem of retiming for latches with multiple clocks and enables. Their solution is to divide latches into classes, where latches share common enables and clocks. Latches can be retimed (forward or backward) across a gate only if they all belong to the same class. The proposed algorithm is correct, but it may not permit all possible retimings. Consider for instance the sub-circuit of Example 1, where the ALU combinational block is fed by latches of two classes with mutually exclusive enables (i.e., whenever the latches of either class are enabled, the latches of the other class are disabled). Forward retiming is forbidden in the original formulation of [6], whereas the retiming of a class of latches would be enabled by the insertion of additional "dummy" latches: This is achieved in Example 1 with the extra register on the opcode input of the ALU block. The above improvement was suggested in [5] for multiple clock phases.

Our implementation follows [6], with the previously cited improvement. Moreover, we support both manual and automatic detection of enables. The latter technique (automatic detection) is derived from [7]. Any next state function δ_i is expressed as

$$\delta_i = e_i \cdot data_i + \overline{e_i} \cdot x_i,$$

where e_i is the enabling function, $data_i$ is the data input, and x_i is the present state. In short, given the BDD of δ_i, we determine e_i and $data_i$ as solutions of the above equation.

4.2 Latches on Primary Input Lines

An additional degree of freedom we have with respect to traditional retiming is the ability to arbitrarily delay or anticipate single primary input signals from their original timing behavior. A given input line could be delayed by the insertion of a dummy latch without enable (always enabled), to be used for forward retiming purposes. This would imply that the associated input signal would be anticipated of one clock cycle, to preserve the required FSM behavior, but strict input–output equivalence would be violated.

A dual effect is produced by artificially removing latches without enable originally present or appeared on input lines as a result of backward retiming. These latches are called *peripheral* input latches in [5], where retiming is targeted to producing as many peripheral latches as possible, to be removed within a simulation run.

Both the above optimizations are possible in our verification framework, where retiming is only a temporary transformation, unrelated to logic synthesis steps. More specifically, latches (without enable) on primary input lines have zero cost for retiming, and dummy latches may be generated whenever useful to make a retime feasible.

Peripheral latches may appear on the output too. In this case the techniques presented in [5, 8] may be considered.

4.3 Transforming Transition Relations

A key issue for BDD based symbolic traversals is supporting partitioned/clustered transition relations, to avoid the BDD blow-up problem often related to their monolithic counterparts.

In our case, this means to compute only partially the transformed transition relation (see Equation (2)), and to keep it in a clustered form. The above transformation consists in some state variable substitutions, latch removals and/or appearance.

- Variable substitution is straightforward (and linear in the BDD size) if the new variables keep the relative ordering of the replaced ones.
- A circuit point, where more latches are chained as a result of retiming, is a source of new next state functions, which are very simple as they are associated to couples of cascaded latches and they can be expressed as

  $$\delta_i = e \cdot x_j + \overline{e} \cdot x_i$$

  (where the j-th latch feeds the i-th one).
- A circuit node, where a retimed latch is removed and no other latch is repositioned, is a function composition point.

We keep the original relative order position for all new variables associated to a circuit node, and we fully solve the first two cases (which imply a negligible time and space effort), whereas we do not explicitly perform function compositions associated to latch disappearance. We address the latter problem by means of auxiliary variables, considered as pseudo-inputs or cut-point variables, fully integrated (and quantified out) in the following verification tasks: Transition relation clustering and image/preimage computations. We operate variable substitution and auxiliary variable generation on the fully partitioned transition relation ($\mathsf{TR} = \prod_i (y_i \equiv \delta_i)$).

5 Experimental Results

The experiments we present here are limited to reachability analysis, as a first and general experimental setup, unrelated from the verification of specific properties. Our main goal is to prove that the sequential behavior of the circuits presented can be analyzed with relevant improvements by using retiming. Some of the advantages we attained could be (partially or totally) denied by verifying properties related to retimed latches.

The presented technique is implemented within an home-made reachability analysis tool, built on top of the Colorado University Decision Diagram (CUDD) package.

We compare traversal performance on the original circuits and on the retimed ones. In all the cases we try to optimize performance using appropriate settings (clustering threshold, variable reordering threshold, etc.). The initial variable ordering is a good one, computed either manually or automatically. We usually enable dynamic reordering with the default settings of the CUDD package.

Our experiments ran on the following machines: a 266 MHz Pentium II with a 384 Mbyte main memory, a 400 MHz Pentium II with a 128 Mbyte main memory, and a 400 MHz Pentium II with a 1 Gbyte main memory, all running RedHat Linux. To simplify direct comparison we have normalized all CPU times to the fastest machine, with a normalization factor of 0.7 (derived experimentally) for the 266 MHz machine.

We present data for a few ISCAS'89, ISCAS'89-addendum benchmarks, and some other circuits [9]. They have different sizes, within the range of circuits manageable by state-of-the-art reachability analysis techniques. We only report here data for the circuits we could retime and traverse with some gain. The benchmark circuits we tried without any significant result are: s1269, s1423, s1512, s3330, s3271. Our technique did not reduce the number of memory elements of circuits s1423, s1512, and s3271, practically leaving the circuits unchanged. In the case of circuits s1269, and s3330, the number of memory elements was reduced, but with slight improvement in the efficiency of reachability analysis.

Table 1 collects statistics on the circuit used, and the retiming process.

For each circuit it shows the number of primary inputs, # I, and some statistics before and after retiming. # L indicates the number of latches, D the sequential depth, and States the final number of reached states. We present here partial reachability analysis results, as on both the circuits, original and retimed, we were unable to complete the analysis.

Palu [9] is a pipelined circuit with an ALU and a register file. At each clock cycle the pipeline starts the execution of an instruction, which completes in three cycles unless stalled: Read the operands from the register file, perform the ALU operation, write result back to the register file. The pipeline supports bypass of the write-back stage. Therefore, if an instruction depends on the result of the one immediately preceding it, the pipeline needs to stall for just one cycle.

Circuit	# I	Original Circuit			Retimed Circuit		
		# L	D	States	# L	D	States
s3384	43	183	–	–	147	–	–
s4863	49	104	4	$2.19 \cdot 10^{19}$	96	3	$2.20 \cdot 10^{18}$
$s5378_{164}$	35	164	44	$3.17 \cdot 10^{19}$	131	44	$2.21 \cdot 10^{16}$
$s5378_{179}$	35	179	44	$3.17 \cdot 10^{19}$	144	43	$4.39 \cdot 10^{16}$
$Palu_8$	17	165	–	–	88	10	$7.77 \cdot 10^{25}$
vsaR	17	66	36	$1.62 \cdot 10^{14}$	62	36	$2.89 \cdot 10^{14}$
$Rotator_{16}$	20	32	2	$1.00 \cdot 2^{32}$	32	2	$1.00 \cdot 2^{32}$
$Rotator_{32}$	37	64	2	$1.00 \cdot 2^{64}$	64	2	$1.00 \cdot 2^{64}$

Table 1: Retiming for reachability analysis. – means data not computed.

vsaR [9] is a very simple microprocessor with no pipelining and no interrupts. The instruction set is limited to basic operations (load, store, add, sub, etc.). All instructions are specified by a 12 bit operating code. There are 3 general-purpose 5-bit registers that can act as source or destinations for the various instructions. All instructions execute in exactly 5 clock cycles. The program counter has only 5 bits to reduce the sequential depth of the FSM.

Rotator [9] has two stages. An input register is fed by primary inputs. An output register stores a rotated copy of the inputs register. The number of rotated bits is determined by a control value of five bits. Every bit of the output register depends on every other bit of the input register. All states are reachable.

More specifically, s3384, s5378, and Palu circuits show relevant reductions of number of latches, moderate reductions are attained with s4863 and vsaR. No reduction is presented by Rotator.

Table 2 collects reachability analysis statistics. For each circuit (retimed and not), it shows the size (in terms of number of BDD nodes) of the largest (R. Peak), and the final (R. Final) reachable state set, the largest partial product created during image computations (BDD Peak), the memory used by the entire process (Mem., in Mbytes [Mb]), and the CPU time (in seconds [sec]).

The advantages obtained on the retimed circuits are evident both in terms of memory and of CPU time.

The traversal of the retimed circuit s3384 was stopped at level 100 for time limit. We know from a guided search experiment that the sequential depth of the circuit has an upper bound of 552. Since our traversal data showed a counter-like behavior we expect we could complete the experiment with a larger time limit.

For circuit vsaR the larger number of reachable states in the retimed version is due to the insertion of a 3 bit dummy register before retiming. A fair comparison would imply accounting this register in the original circuit. Both versions of s5378 and Palu show relevant memory and CPU time gain. In particular the Palu example shows that retiming can be very effective with pipelining

Circuit	Original Circuit					Retimed Circuit				
	R. Peak	R. Final	BDD Peak	Mem. [Mb]	Time [sec]	R. Peak	R. Final	BDD Peak	Mem. [Mb]	Time [sec]
$s3384_{level\ 6}$	164343	138276	433623	172	3058	114395	74750	321323	164	2513
$s3384_{level\ 100}$	–	–	–	–	–	485242	314395	672634	195	475772
s4863	8241	8241	17709	14	24	5183	5183	18160	14	21
$s5378_{164}$	125326	46981	761569	55	29997	35324	24763	145421	26	3402
$s5378_{179}$	376219	117086	992340	67	25283	84928	62746	464937	36	6752
$Palu_8$	–	–	–	–	–	101857	756	406186	35	625
vsaR	4540095	335694	8524412	648	6975	2478336	49494	4029557	288	1148
$Rotator_{16}$	17	1	111263	8	1	17	1	33	1	0
$Rotator_{32}$	–	–	–	–	–	33	1	65	1	0

Table 2: Reachability analysis comparison. – means data not computed for memory overflow.

units.

Rotator is a very particular case. With this circuit a backward retiming of the output register across the rotating logic makes reachability a trivial task (an input register directly feeding an output register with no intermediate logic). The 32 bit version of the original circuit could not be completed with the available resources. We should expect a verification performance ranging between the two extreme cases depending on the verified property.

6 Conclusions and Future Work

We present a new approach to improve reachability analysis, and possibly all verification problems based on reachability analysis.

We exploit retiming as a property preserving transformation to reduce the number of latches of a FSM, and to reposition them to obtain a simplified state set representation. In this framework retiming is seen as a temporary transformation to be applied to the FSM, and to the property as well.

Experimental results on benchmark and home made circuits show lower image computation costs, and reduced state set representation sizes.

Future works will investigate new heuristics, and optimality criteria for retiming to further improve reachability analysis efficiency. Moreover, we will move toward property verification, e.g., model-checking, by transforming properties as well.

References

[1] D. Stoffel and W. Kunz. Record & Play: A structural Fixed Point Iteration for Sequential Circuit Verification. In *Proc. IEEE/ACM ICCAD'97*, pages 394–399, San Jose, California, November 1997.

[2] G. P. Bischoff, K. S. Brace, S. Jain, and R. Razdan. Formal Implementation Verification of the Bus Interface Unit for the Alpha 21264 Microprocessor. In *Proc. IEEE ICCD'97*, Austin, Texas, October 1997.

[3] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton. Using Combinational Verification for Sequential Circuits. In *Proc. IEEE/ACM DATE'99*, pages 138–144, Munich, Germany, March 1999.

[4] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model Checking the IBM Gigahertz Processor: An Abstraction Algorithm for High-Performance Netlists. In *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 72–83, Berlin, 1999. Springer-Verlag. LNCS 1633.

[5] A. Gupta, P. Ashar, and S. Malik. Exploiting retiming in a guided simulation based validation methodology. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 350–353, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[6] C. Legl, P. Vanbekbergen, and A. Wang. Retiming of Edge–Triggered Circuits with Multiple Clocks and Load Enables. In *IWLS'99: IEEE International Workshop on Logic Synthesis*, May 1999.

[7] K. Ravi. Improvements to Reachability Analysis. Unpublished manuscript, October 1997.

[8] G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large Finite State Machine. In *Proc. IEEE/ACM ICCAD'96*, pages 354–360, San Jose, California, November 1996.

[9] K. Ravi and F. Somenzi. Hints to Accelerate Symbolic Traversal. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 250–264, Berlin, September 1999. Springer-Verlag. LNCS 1703.