# Latch Optimization in Circuits Generated from High-level Descriptions*

Ellen M. Sentovich, Horia Toma, Gérard Berry
Ecole Nationale Supérieure des Mines de Paris
Centre de Mathématiques Appliquées
06904 Sophia-Antipolis, FRANCE

*In a gate-level description of a finite state machine (FSM), there is a tradeoff between the number of latches and the size of the logic implementing the next-state and output functions. Typically, an initial implementation is generated via explicit state assignment or translation from a high-level language, and the tradeoff is subsequently only lightly explored. We efficiently explore good latch/logic tradeoffs for large designs generated from high-level specifications. We reduce the number of latches while controlling the logic size. We demonstrate the efficacy of our techniques on some large industrial examples.*

## 1 Introduction

In a gate-level description of a finite state machine (FSM), there is a tradeoff between the number of latches and the size of the logic implementing the next-state logic. This tradeoff can be exploited at two levels: during generation of the initial implementation, and during subsequent logic optimization steps.

### 1.1 Background

*State assignment* is the generation of a state encoding and an initial latch/logic implementation from a higher level in the design process. To date, primarily two approaches have been used. *Explicit state assignment* begins from an explicit state transition graph and chooses a minimum-latch encoding while minimizing the size of the combinational logic [5, 13, 6]. *State assignment from high-level languages* chooses an encoding according to the delay statements in the specification, relying on logic synthesis to later optimize the gate-level implementation [1]. Explicit state assignment is impractical for large designs, and despite sophisticated techniques for determining an optimal assignment, it can produce results far worse than hand-coded implementations. Furthermore, explicit state assignment programs have not targeted greater-than-minimum-latch implementations. With current technology (e.g. FPGAs), it is no longer necessary to minimize the number of latches and doing so often produces prohibitively large combinational logic. One-hot encoding can also be applied to an explicit state graph, where one latch is used for each state. The resulting logic will be small and fast since the states do not need to be encoded and decoded. However, the number of latches is huge, and a one-hot implementation can be a difficult starting place for logic synthesis. Automatic techniques for reducing the number of latches in a one-hot implementation to produce a good tradeoff have not been resoundingly successful.

State assignment from high-level languages is typically done by a statement-by-statement translation, which results in a natural insertion of registers at the delay statements in the description. This is a good starting point for logic synthesis, but results in far more latches than are required to implement the design [1]. Even if the number of latches is not important for the final implementation, too many can drastically reduce the efficiency of synthesis and optimization algorithms (e.g., symbolic state traversal).

After state assignment, the latch/logic tradeoff can be explored via logic optimization. Standard techniques, e.g., extracting common factors, function simplification, and retiming, may result in some latch/logic rearrangement, but largely the state assignment is fixed and optimization only improves the implementation for the given assignment. The goal of retiming and resynthesis [8] was to jointly optimize latch positions and combinational logic, but the optimization potential was too limited by the initial state assignment.

Removal of redundant latches has been done ubiquitously. The term "redundant latches" itself is used ubiquitously with a variety of meanings: latches that do not fanout to a primary output, latches that are constant in the entire state space, latches that are constant in the reachable state space, latches that are equivalent to other latches, and other more sophisticated definitions. We do not consider simple redun-

*This work was supported in part by the National Science Foundation under grant INT-9505943, and the French GENIE MESR INRIA project.

dant latch removal here (latches that do not fanout, latches that are constant) as the results are straightforward. Equivalent state variable identification and removal has been done as part of several works (e.g., [10, 12, 9]). With current BDD and symbolic traversal techniques, this also is a simple operation. A more robust algorithm for single latch removal was published in [7]. It is directly relevant to our single latch removal algorithm described in Section 3.1 and is treated more thoroughly in that section. It is exact (precisely for replacing a single latch with logic), and far too expensive when the number of latches exceeds the minimum ($log_2(reachable-states)$) even for small circuits. In [9], a more general technique for re-encoding finite state machines was given. While some of the work there is relevant to ours (e.g., controlling the size of the encoding/decoding logic), the target was quite different. The goal of that re-encoding was to try to match the encodings of two different but similar machines in order to speed up symbolic traversal.

### 1.2 Our Approach

A complete exploration of the latch/logic tradeoff is certainly not feasible or necessary. Nonetheless, a sufficiently rich choice of solutions should be available, especially with the variety of implementations found in today's technology. For example:

- one may target an implementation in hardware or software
- a hardware implementation may be targeted to a final custom design or an FPGA board for emulation
- the current specification may be a preliminary one used only for verification of functional properties.

In the above three cases, the optimization criteria are quite different.

We focus on efficient exploration of the latch/logic tradeoff for a design generated from a high-level specification. In particular, we begin with designs generated from Esterel descriptions. The initial encoding (generated by the Esterel compiler), while a natural one with respect to the behavior of the design, contains many redundant registers. We develop algorithms for removing redundant registers. We target the algorithms to work well on large designs, and to remove registers as cheaply as possible (i.e., to easily discover redundant registers, to be able to easily replace them with a minimum amount of additional logic). We generate optimal solutions considering final implementation cost and/or efficacy of the intermediate representations.

Our key results include the following:

- Our algorithms are efficient enough to provide a significant choice of implementations regarding the latch/logic tradeoff for very large circuits. No other results in this direction have been published thus far that we are aware of.
- By applying *simple* (easy to compute) heuristics for latch reduction, we get very close to the *minimum* number of latches possible on a given reachable state set. In almost all our examples we obtained a final number of latches less than or equal to $log_2|R| + 1$, where R is the set of reachable states.
- Because we work incrementally from an initial implementation taken from a high-level language, and we strive to preserve the given structure, the size of the resulting logic remains tractable even as we approach the minimum number of latches.
- Our results for maximum latch removal, in terms of area of the resulting logic, compare very favorably to those obtained by the traditional "robust" technique of extracting the state transition graph, running an explicit state encoding program, and performing logic optimization on the result: we usually obtain a smaller implementation in much less time. Furthermore, we observed a blow-up in the area of the initial implementation after state assignment (resulting in very long logic optimization run-times) which we do not experience with our incremental techniques. We could only make this comparison on relatively small examples, as the explicit state transition graph is expensive to compute.
- Run-times for the latch removal algorithms are insignificant. The reachable state set is computed initially, and all subsequent latch redundancies are determined in a comparatively trivial amount of time from this set.

## 2 Overview of the Technique

We apply heuristic techniques to successively remove redundant latches while controlling the size of the combinational logic. A complete latch removal technique could be implemented by extracting an explicit state transition graph, performing exact state minimization, and running a state assignment program to generate a minimum-latch encoding. This is far too expensive and yields no insight to the latch/logic tradeoff. We instead successively remove latches using a *subset* of the information that is used in the aforementioned complete scheme. (For example, we consider reachable states, but not equivalent states.)

### 2.1 The General Algorithm

The core algorithm proceeds as follows:

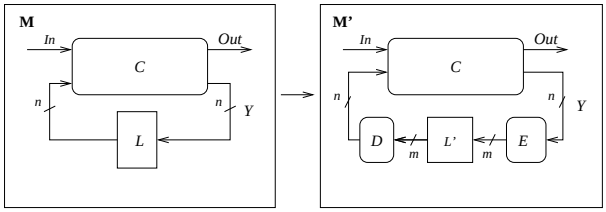



Figure 1: General Circuit Transformation

1. Compute the reachable states of the machine.
2. Determine a set of latches which may be *easily* removed while *preserving the reachable state set.*
3. Among these, choose a set of latches to remove based on a cost estimate; remove latches, add logic, and optimize.
4. Iterate, removing as many latches as possible, or as many latches as desired given constraints on the size of the combinational logic.

It is important to note that at each one-latch-removal step, for each of our proposed algorithms, we re-encode states by considering state pairs which can be *easily merged.* This is not a general re-encoding: we pair each reachable state with an unreachable state, and give each pair a single new encoding. The pair/merge operation is done in such a way that it is computationally easy to determine the pairings, it is inexpensive in BDD size and logic size to perform the mergings and update the reachable states, and the existing logic is preserved as much as possible as it contains valuable information on the structure of the circuit. Thus the computations required remain tractable for our very large circuits, and the resulting additional combinational logic (which is usually the most limiting factor in incremental re-encoding algorithms) is controllable.

The transformation is proven to preserve behavior on the reachable state set and remains correct for every over-approximation of the reachable state set. Therefore it could be used in conjunction with efficient techniques for approximate reachability analysis [3].

### 2.2 Circuit Transformation

The transformation is illustrated in Figure 1. The original FSM is $M$ and the transformed one is $M'$. We call $L = \{l_k \mid 1 \leq k \leq n\}$ the set of latches of $M$. A transformation will remove latches with indices in a set $I$. For convenience, we keep the same indices for the remaining latches in $M'$: $L' = \{l'_k \mid 1 \leq k \leq n,\ k \notin I\}$, and we assume that $L'$ has $m$ latches. The next state vector for $L$ in $M$ is generated by $C$ and called $Y$. In $M'$, the *encoding function* $E$ has type $B^n \rightarrow B^m$ and it is given by a vector of functions $E_k : B^n \rightarrow B$, $1 \leq k \leq n,\ k \notin I$. The *decoding function* $D$ has type $B^m \rightarrow B^n$ and is given by a vector $D_k : B^m \rightarrow B$, $1 \leq k \leq n$.

Let $R \subseteq B^n$ (resp. $R' \subseteq B^m$) be the set of reachable states in $M$ (resp $M'$), and let $\mathcal{R} : B^n \rightarrow B$ (resp. $\mathcal{R}' : B^m \rightarrow B$) denote the characteristic function of $R$ (resp. $R'$). Thus $r \in R$ implies $\mathcal{R}(r) = 1$. Let $r_0$ and $r'_0$ denote the initial states of $M$ and $M'$.

In this context we say that $M$ is equivalent to $M'$ if $r_0 = D(r'_0)$ and $D(E(r)) = r$ for all $r \in R$. This is the property our transformations will satisfy.

Functions will be represented by polynomials or BDDs. The input variables will be consistently called $y_k$ for $E$, $l'_k$ for $D$, $l_k$ for $\mathcal{R}$, and $l'_k$ for $\mathcal{R}'$. If $F$ denotes a polynomial or BDD over a set of variables $X$ and if $Y$ is another set of variables where $|X| = |Y|$, we denote by $F[Y/X]$ the result of the substitution of the $x_k$ by the $y_k$, and $F[\overline{Y}/X]$ the result of the substitution of the $x_k$ by the complements $\overline{y_k}$. We respectively denote by $F_x$ and $F_{\bar{x}}$ the positive and negative cofactor of $F$ with respect to $x$.

## 3 Algorithms

In this section, several algorithms for latch removal are described. In summary:

1. `single-latch` removal: determine which latches can be removed individually and replaced by a combinational function of the other latches.
2. `2-by-1` : determine pairs of latches that can be removed and replaced by a single latch whose input is a combinational function of the other latches.
3. `n-by-(n-1)` : replacement of $n$ latches by $n-1$ latches with a new combinational function for each latch.

Each is specified according to

- the *condition* under which latches are removed,
- the logical *transformation* required in the circuit, i.e. the specification, of $D$ and $E$ in Figure 1,
- a brief description of the *algorithm* (proofs are omitted).The new initial state is trivially computed from the encoding function $E$ and hence not discussed further.

### 3.1 Transformation `single-latch`

**Condition:** A single latch $l_i$ can be replaced by a combinational function of the others if

$$\mathcal{R}_{l_i} \cdot \mathcal{R}_{\bar{l}_i} = 0 \tag{1}$$

This condition was originally given in [2]. Its satisfaction implies that $l_i$ does not distinguish any reachable states; the re-encoding will couple each reachable

state of the form $l_1 l_2 \ldots l_{i-1} 0 l_{i+1} \ldots l_n$ with the unreachable state $l_1 l_2 \ldots l_{i-1} 1 l_{i+1} \ldots l_n$ to produce the state $l'_1 l'_2 \ldots l'_{i-1} l'_{i+1} \ldots l'_n$ and similarly for 1/0.

In general, a subset of the latches will each satisfy this condition. Once a single latch is removed, the remaining subset of removable latches may change. We apply heuristic techniques, as described in the algorithms below, to determine which latches to remove.

In [7], an exact branch-and-bound algorithm is used to determine the maximum number of single latches that can be removed. This algorithm is far too expensive, and we have empirically observed results that nearly match the `single-latch` algorithms with it. Furthermore, we reduce the number of latches even further with the algorithms described in the sequel.

**Transformation:** The latch $l_i$ is removed and the logic for functions $D$ and $E$ are added, where $E$ is defined by $E_k(Y) = y_k$ for $k \neq i$. For $D$, we set $D_k(L') = l'_k$ for $k \neq i$ and

$$D_i(L') = \mathcal{R}_{l_i}[L'/L] \tag{2}$$

**Algorithm :** The pseudo-code for the algorithm is shown in Figure 2. It greedily selects and removes one latch at a time based on a cost function related to the potential for removing other latches. We use the branching heuristic of [7], so `cost()` sets $C_{l_i} = |\mathcal{R}| - abs(|\mathcal{R}_{l_i}| - |\mathcal{R}_{\bar{l}_i}|)$, where $|\mathcal{R}|$ is the onset size of the BDD $\mathcal{R}$. The absolute value term is highest for those latches with the most potential for distinguishing states. By selecting the latch with the lowest cost $C_{l_i}$, we leave the latches with the highest potential, and thus greedily maximize the chances of removing more latches. Furthermore, this heuristic implies a minimum number of minterms that are changed in the encoding space, which we observe to help control the size of the overall logic. After selecting the latch, $\mathcal{R}$ is updated as though the latch had already been removed, and the process is iterated. After a set of latches that can be removed simultaneously have been computed, they are removed and replaced by combinational logic that depends only on the remaining latches. This algorithm computes a maximal removable set, and hence iteration is not necessary.

We have also used a cost function based on the size of the support of the BDD for $D_i$. This is not a tight measure of implementation size, but there is a correlation. In this case `is_removable()` is modified so that a latch is removed only if the support of the BDD is less than a given bound. Typically the algorithm is iterated while relaxing the bound.

Note that upon removing the latch, the output variable implementing the input of the latch

```
single-latch(M, R(L))
{
     R_temp = R;
     removed_list = φ;
     /* Find and remove latches.  */
     while (1) {
          best_latch_cost = ∞;
          foreach latch l_i {
               if is_removable(R_temp, l_i) {
               /* By condition (1) */
                    C_{l_i} = cost(l_i, R_temp);
                    if (C_{l_i} < best_latch_cost) {
                         best_latch_index = i;
                         best_latch_cost = C_{l_i};
                    }
               }
          }
          if (best_latch_cost ≡ ∞) break;
          i = best_latch_index;
          R_temp ← ∃_{l_i} R_temp;
          removed_list ← removed_list ∪ l_i;
     }
     if (removed_list ≡ φ) return;
     /* Compute D and modify M.  */
     R_new = R;
     foreach l_i ∈ removed_list {
          D_i = R;
          foreach l_j ∈ removed_list, j ≠ i {
               D_i ← ∃_{l_j} D_i;
          }
          D_i ← D_{i l_i}[L'/L]
          /* By transformation (2) */
          add_logic(M, D_i);
          remove_latch(M, l_i = 0);
          R_new ← ∃_{l_i} R_new;
     }
     R ← R_new;
}
```

Figure 2: `single-latch` removal

is discarded, which results in simplification of the combinational logic $C$ by sweeping. In addition, each $D_k$ is computed directly from the initial $\mathcal{R}$ by smoothing all the other variables that will be removed: if $l_1, l_2, \ldots l_j$ are simultaneously removable, $D_1 = (\exists_{l_2}, \exists_{l_3}, \cdots \exists_{l_j} . \mathcal{R}_{l_1})[L'/L]$, and similarly for $D_2, D_3, \cdots, D_j$. Furthermore, the operations can be performed in any order since they commute. Alternately, $\mathcal{R}$ could be updated as each variable is removed, and the next $D_i$ computed from the updated $\mathcal{R}$. This latter technique requires less computation at each removal, but creates functions that depend upon variables that will be eventually removed and hence increases the levels of logic. As we strive to control size and depth of the logic and as computation time is not significant, we prefer the former method.

### 3.2 Transformation 2-by-1

**Condition:** Two latches $l_i$ and $l_j$ can be replaced by a single latch $l'_j$ if

$$\mathcal{R}_{l_i l_j} \cdot \mathcal{R}_{\bar{l}_i \bar{l}_j} + \mathcal{R}_{l_i \bar{l}_j} \cdot \mathcal{R}_{\bar{l}_i l_j} = 0 \tag{3}$$

The satisfaction of this condition implies that there is again a valid pairing of reachable states and unreach-

able states. The re-encoding will couple each reachable state $l_1 l_2 \ldots l_{i-1} 0 l_{i+1} \ldots l_{j-1} 0 l_{j+1} \ldots l_n$ with the unreachable state $l_1 l_2 \ldots l_{i-1} 1 l_{i+1} \ldots l_{j-1} 1 l_{j+1} \ldots l_n$ to produce $l'_1 l'_2 \ldots l'_{i-1} l'_{i+1} \ldots l'_{j-1} 0 l'_{j+1} \ldots l'_n$. That is, $l_i l_j = 00$ or 11 is replaced by $l_j = 0$ and $l_i l_j = 01$ or 10 is replaced by $l_j = 1$.

**Transformation:** If $l_i$ and $l_j$ satisfy the above condition, one can remove $l_i$ and set

$$\begin{cases} E_j(Y) = y_i \cdot \bar{y}_j + \bar{y}_i \cdot y_j \\ E_k(Y) = y_k, \quad k \notin \{i, j\} \end{cases} \tag{4}$$

$$\begin{cases} D_k(L') = l'_k, \quad k \notin \{i, j\} \\ D_i(L') = (l_j \cdot \mathcal{R}_{l_i \bar{l}_j} + \bar{l}_j \cdot \mathcal{R}_{l_i l_j})[L'/L] \\ D_j(L') = (l_j \cdot \mathcal{R}_{\bar{l}_i l_j} + \bar{l}_j \cdot \mathcal{R}_{l_i l_j})[L'/L] \end{cases} \tag{5}$$

**Algorithm:** Each latch pair is examined and replaced by a single latch if possible. Note that in this case, the entry $y_i$ of the removed latch $l_i$ cannot be discarded, because $E_j$ depends on it. Therefore, there is no subsequent logic reduction in $C$.

### 3.3 Transformation `n-by-(n-1)`

The `n-by-(n-1)` algorithm considers the entire encoding space when searching for a state-pair merging, rather than restricting to merging across a plane or a small cube (as the case for `single` and `2-by-1`). The algorithm has two parts. First, the encoding is modified (without removing latches) by clustering the existing encodings toward the all-0 encoding. This is called `migrate-states`. Next, the resulting encoding is checked to see if each reachable state can be paired with its mirror state (all variables complemented).

#### 3.3.1 Transformation `migrate-states`

**Condition:** The condition that some encodings can be shifted toward the origin is given by

$$\exists l_i \;\; s.t. \;\; \mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}} \neq 0 \tag{6}$$

The reachable states in $l_i \cdot \mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}}$ are re-encoded as $\bar{l}_i \cdot \mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}}$. Since $\bar{l}_i \cdot \mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}} \not\subseteq \mathcal{R}$, this transformation safely maps a state in the reachable set to an unused encoding (in the unreachable set).

**Transformation:** Given $l_i$ that satisfies (6), the machine is re-encoded as follows:

$$\begin{cases} E_j(Y) = y_j, \quad j \neq i \\ E_i(Y) = (l_i \cdot \mathcal{R}_{l_i} \cdot \mathcal{R}_{\bar{l}_i})[Y/L] \end{cases} \tag{7}$$

$$\begin{cases} D_j(L') = l'_j, \quad j \neq i \\ D_i(L') = l'_i + (\mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}})[L'/L] \end{cases} \tag{8}$$

**Algorithm:** The encodings are moved in a greedy fashion towards the origin in the Boolean space. The algorithm is implemented simply by iterating over the latches, checking the condition and performing the transformations if possible. The motivation is that the subsequent `fold-states` operation can be performed if all the reachable states are Hamming distance $\lceil \frac{n}{2} \rceil - 1$ of the origin[1]. We could choose any point in the Boolean space around which to cluster the encodings, but the all-0 encoding is a good choice for Esterel circuits (and, we believe, for others generated from high-level descriptions). The reason is that, while the initial encodings are not one-hot, they are close to one-hot, and "group-hot"; as such they contain many 0's. `migrate-states` can add an exorbitant amount of logic, so rather than iterating to completion, we check the `fold-states` condition at each iteration and stop when it is satisfied. The reachable states are updated after each latch is visited as follows:

$$\mathcal{R}' = l'_i \cdot (\mathcal{R}_{l_i} \cdot \mathcal{R}_{\bar{l}_i})[L'/L] + \bar{l}'_i \cdot (\mathcal{R}_{\bar{l}_i} + \mathcal{R}_{l_i} \cdot \overline{\mathcal{R}_{\bar{l}_i}})[\overline{L'}/L]$$

#### 3.3.2 Transformation `fold-states`

The re-encoding is done by choosing a latch $l_i$, and for each state encoding, if $l_i = 1$ the encoding is unchanged, while if $l_i = 0$ the other state variables are inverted. The latch $l_i$ is then removed. For example, if $R = \{000, 110, 010, 011\}$ the condition is satisfied. Choosing the first bit for $l_i$, the new encoding is $R' = \{11, 10, 01, 00\}$.

**Condition:** The condition under which the states can be merged is given by

$$\mathcal{R}[\bar{L}/L] \cdot \mathcal{R} = 0 \tag{9}$$

**Transformation:** The transformation removes register $l_i$ and sets

$$E_j(Y) = \bar{y}_i \cdot \bar{y}_j + y_i \cdot y_j, \quad j \neq i \tag{10}$$

$$\begin{cases} D_i(L') = \mathcal{R}_{l_i}[L'/L] \\ D_j(L') = l'_j \cdot D_i + \bar{l}'_j \cdot \bar{D}_i, \quad j \neq i \end{cases} \tag{11}$$

**Algorithm:** The implementation follows directly from (10) and (11). The reachable states are updated as follows:

$$\mathcal{R}' = \mathcal{R}_{l_i}[L'/L] + \mathcal{R}_{\bar{l}_i}[\overline{L'}/L]$$

### 3.4 Comments on the Algorithms

It must be emphasized that in selecting algorithms and heuristics, we do not focus primarily on traditional logic optimization metrics. Our goals while exploring the latch/logic tradeoff are instead to

- maintain the initial existing logic structure to the extent possible, as it reflects the structure given by the high-level description,

[1] This is a sufficient but not necessary condition.

- use metrics that relate to the perform of our algorithms and to the size of the $D$, $E$ logic which we have the most control over: we try not to overly pessimize logic synthesis,
- leave the logic optimization to existing tools that are specialized for this purpose.

**Heuristics:** The most important heuristic that we have not described is related to don't care conditions for selecting $D$ and $E$. In each case, there is actually a set of combinational functions that can be used, not just a single one. The set arises from the use of the unreachable states as don't care conditions; we have not indicated this choice in our description of the $D$ and $E$ functions. For example, in the `single-latch` algorithm, any function that satisfies $\mathcal{R}_{l_i}[L'/L] \subseteq D_i(L') \subseteq (\mathcal{R}_{l_i} + \overline{\mathcal{R}_{\bar{l}_i}})[L'/L]$ would be correct. We experimented with different choices, and found that the functions were small enough that this degree of flexibility was not useful at this level. Furthermore, since it arises solely from the reachable state set, the same information can be used instead in subsequent logic optimization.

There are many other heuristics that can be employed for selecting a latch, for minimizing the BDDs, for minimizing the implementation, for optimizing algorithm performance, etc. We focus on finding good implementations and exploring a reasonable subset of the latch/logic tradeoff given the available tools (state-of-the-art BDD technology, logic optimization, etc.) for the designs in our domain, rather than attempting to implement any exact algorithms or thoroughly test a large set of heuristics whose final value is difficult to measure. We tested a number of heuristics (especially for selecting $D$ and $E$) and our choices in function implementation reflect the results of these experiments.

**Other similar algorithms:** Note that the `2-by-1` is a generalization of the single latch removal algorithm, which can be further generalized to replace 3 registers by 2, etc. We found that such a successive generalization did not improve the results sufficiently to justify its rapidly increasing cost. (Note that the `n-by-(n-1)` algorithm described above is not a generalization of `2-by-1`).

**Completeness:** The `single_latch` algorithm is complete in that when it is finished, no single latch can be removed while maintaining the same reachable state set on $Y$. Algorithm `2-by-1` is not complete in that it may be possible for two latches to be replaced by one latch with additional combinational logic and not replaced by this algorithm . Similarly, the `n-by-(n-1)` algorithm is not complete.

# 4 Implementation and Results

Experiments have shown that finding a good latch encoding before performing optimization is very important as the encoding strongly effects optimization. We know that a $\log_2 |R|$ encoding usually implies exorbitant combinational logic, but given a particular encoding we cannot predict what the size of the combinational logic will be. The same intuition applies to the tradeoff between the number of latches and the performance of verification algorithms.

The aim of our implementation was to develop a tool which allows us to make estimates over the starting points of combinational optimization for hardware and software designs and for verification. These metrics imply the need for different strategies.

## 4.1 Implementation

We implemented our program `rem_latch` using the TiGeR library [4] for BDDs and reachable states, and the Berkeley SIS environment [10] to perform combinational logic optimization. We used mainly two scripts for logic optimization in SIS: a fast but less robust one (COMBOPT), and a more expensive one (BLIFOPT). Where actual logic cost is estimated, literal count in SIS is computed.

### 4.1.1 Strategy 1: Implementation

The first strategy is oriented to hardware and software *implementations*, where we use `single_latch` with a cost function based on BDD size. The transition from BDDs to logic can be costly, and we found that BDD support size was the best measure for controlling this blow-up. Recall that `single-latch` actually reduces the size of $C$, so the overall logic cost (using the post-synthesis measure of literal count) varies very little as the registers are removed (see Section 4.2).

During the experiments we observed that very attractive configurations can be discovered even for circuits where large intermediate BDDs are generated. In these cases, subsequent logic optimization successfully reduced the implementation sizes. For this reason, we iterate `single-latch` while relaxing the cost conditions, continue with `2-by-1`, allow a logic increase, and optimize later.

### 4.1.2 Strategy 2: Verification

Experiments demonstrate that reducing the number of latches has a positive effect on the performance of verification techniques: the BDDs for the reachable states decrease in size and the combinational logic grows slowly. The reason for this is primarily that the number of latches has a strong effect on the BDD sizes

(there are two BDD variables per latch for FSM verification). Strategy 2 uses `single-latch` to remove the maximum number of latches followed by `2-by-1` iterated to completion. This latter is applied alternatively with logic optimization (BLIFOPT) to control the implementation size. For the largest circuits, we used the COMBOPT script. The logic grew more quickly and consequently we were restricted in the number of latches that were removed. Nonetheless, we were able to reduce latches and improve verification times where we were not able to perform any optimization previously.

#### 4.1.3 Strategy 3: Exploration

The goal of the third strategy was to minimize the number of latches to study the behavior of the algorithms and properties of the final circuits. Interestingly, we were able in almost all cases to reduce the number of latches to $\log_2(|R|)+1$, which gives an indication of the power of our algorithms[2].

#### 4.1.4 Controlling the Encoding/Decoding Logic

We already tailor our algorithms to find state pairs that are easy to merge and re-encode, and thereby minimally modify the reached state set. We also control BDD sizes. We implement the new logic from the BDD generated by TiGeR. The TiGeR package creates logic from BDDs that is linear in the number of BDDs nodes. In some cases, we use the formulas for $D$ and $E$ to generate the logic gates, computing other functions as BDDs and substituting the BDD results into the created logic gates. This technique can increase the number of levels of the circuit, so it must be used with caution.

### 4.2 Results

The run-times for all algorithms are comparable to the reachable state computation and hence are not reported here.

The first set test is the ISCAS-89 sequential test benchmarks, which we used for comparison with the exact single latch algorithm in [7]. The results are shown in Table 1. Only for s444 are the results of the exact algorithm better.

The other benchmarks we used are all synthesized by the Esterel v5 compiler. Some of them are simply test programs, but others are large industrial designs. *tcint*, *renault*, *snecma*, *seq*, and *trappes* are particularly large and interesting examples. We have two possible starting points. The designs generated directly from the Esterel compiler have a manageable initial implementation in terms of encoding and logic, but far too many redundant registers. The other case arises from examples that initially have combinational cycles. The causality analysis program [11] generates an acyclic implementation directly from the BDDs and is thus huge in terms of logic (e.g., *trappes*).

[2]Theoretically, `n-by-(n-1)` is guaranteed to reduce the number of latches to $2 \cdot \log_2(|R|)$, but no further.

| Circuit | states | reg | exact | `single` | min |
|---|---|---|---|---|---|
| s208 | 17 | 8 | 5 | 5 | 5 |
| s298 | 218 | 14 | 12 | 12 | 8 |
| s382 | 8865 | 21 | 18 | 18 | 14 |
| s400 | 8865 | 21 | 18 | 18 | 14 |
| s444 | 8865 | 21 | 17 | 18 | 14 |
| s526 | 8868 | 21 | 19 | 19 | 14 |
| s641 | 1544 | 19 | 14 | 14 | 11 |
| s713 | 1544 | 19 | 14 | 14 | 11 |

Table 1: `single-latch` vs exact single removal

| | | | | initial | | `rem_latch` | |
|---|---|---|---|---|---|---|---|
| Circuit | I/O | states | min | reg | lit | reg | lit |
| abc | 4/12 | 16 | 4 | 13 | 239 | 4 | 173 |
| abcdef | 7/24 | 128 | 7 | 25 | 476 | 8 | 485 |
| ctrl | 10/8 | 24 | 5 | 20 | 364 | 5 | 1025 |
| ctrlct | 10/8 | 211 | 8 | 23 | 404 | 9 | 1323 |
| renault | 23/166 | 257 | 9 | 66 | 2022 | 9 | 4253 |
| runner | 6/5 | 5182 | 13 | 30 | 362 | 14 | 972 |
| seq | 55/98 | 109415 | 17 | 121 | 1790 | 60 | 1021 |
| snecma | 23/5 | 10241 | 14 | 70 | 1705 | 14 | 1676 |
| tcintnct | 19/20 | 310 | 9 | 90 | 1036 | 26 | 328 |
| trappes | 53/154 | 135718 | 18 | 157 | 44900 | 20 | 1193 |
| ww | 8/99 | 41 | 6 | 35 | 1098 | 6 | 435 |

Table 2: Initial version vs Minimum-latch version

In Table 2, the initial circuit is compared with the minimum-latch `rem_latch` result optimized with COMBOPT. We obtained close to the minimum number of latches on most examples.

In Table 3 we report our minimum-latch/BLIFOPT results and those obtained from a combination of state-graph extraction, exact state minimization, state assignment with NOVA, and logic optimization in SIS with both the SIS rugged script and BLIFOPT. Our results compare favorably.

In Table 4, we compare the best logic optimization results obtained from applying BLIFOPT to the initial circuit to the best `rem_latch` results obtained with a combination of latch removal and optimization.

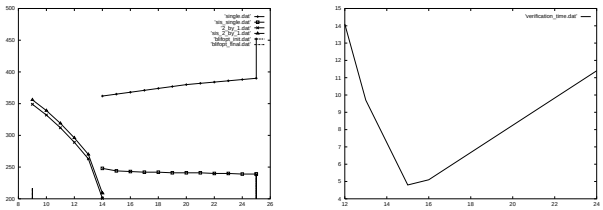
For all small examples, we were able to reduce the

| | `rem_latch` | | NOVA/SIS | | |
|---|---|---|---|---|---|
| Circuit | reg | literals | reg | literals | states |
| abc | 4 | 111 | 4 | 197 | 16 |
| abcdef | 8 | 330 | 7 | 22197 | 128 |
| ctrl | 5 | 626 | 5 | 172 | 23 |
| ctrlct | 10 | 717 | 5 | 167 | 23 |
| tcintnct | 26 | 313 | 8 | 7692 | 231 |
| traffic | 5 | 49 | 5 | 77 | 18 |
| ww | 6 | 250 | 6 | 777 | 41 |

Table 3: `rem_latch` minimum-latch vs NOVA/SIS

| | | | BLIFOPT | | remlatch | |
|---|---|---|---|---|---|---|
| Circuit | in | out | reg | lit | reg | lit |
| abc | 4 | 12 | 13 | 114 | 4 | 111 |
| abcdef | 7 | 24 | 25 | 227 | 9 | 215 |
| ctrl | 10 | 8 | 20 | 142 | 16 | 135 |
| ctrlct | 10 | 8 | 18 | 143 | 14 | 140 |
| renault | 23 | 166 | 37 | 507 | 28 | 497 |
| runner | 6 | 5 | 29 | 198 | 15 | 198 |
| snecma | 23 | 5 | 36 | 491 | 21 | 407 |
| tcint | 19 | 20 | 50 | 241 | 38 | 237 |
| tcintnct | 19 | 20 | 47 | 197 | 38 | 194 |
| ww | 8 | 99 | 13 | 220 | 6 | 217 |
| seq | 55 | 98 | — | — | 60 | 1021 |
| trappes | 53 | 154 | — | — | 20 | 1193 |

Table 4: BLIFOPT vs best_remlatch



a) Latches vs literals b) Verification times

Figure 3: Design evolution during latch removal

number of latches to the minimum so thorough exploration was possible. Here, we tried many combinations of latch removal and optimization, and the algorithms and strategies discussed in Sections 3 and 4.1 reflect this experience. On larger files we couldn't obtain the minimum (due to the size of the encoding logic and not theoretical limits of the algorithms). Instead, we used strategies similar to those that were successful on small examples. The number of literals is comparable, despite the fact that `rem_latch` must add encoding and decoding logic. Furthermore, the number of latches is much lower. For *seq* and *trappes*, no significant logic optimization can be done without first removing latches, so we present novel results on these examples.

The graph in Figure 3a shows the general evolution of the latch-literal tradeoff during the application of the standard strategies (all of our examples behaved similarly): `single-latch` successively removes latches and slightly decreases logic size, a logic optimization step is performed (the discontinuity) and finally `2-by-1` removes latches at a clear expense in logic size. The graph in Figure 3b shows the general evolution of the CPU time for self-verification as the number of latches decreases. The best point is obtained after `single` and `2-by-1`, but without iterating `2-by-1` to completion.

## 5 Future Work

The first goal is to use the results of this work as a pre-processor to logic optimization. This is critical since the circuits produced by causal analysis (or BDDs) are very large and difficult to cope with.

The next step is to exploit particular properties that can be gleaned from high-level specifications and to use this information earlier in the process, i.e., remove latches before computing the full reachable state set. It is important with large designs to exploit and preserve the given natural circuit structure.

For hardware implementations using FPGAs, we will explore *increasing the number of latches* via retiming to improve the critical path.

## References


[1] G. Berry and H. Touati. Optimized Controller Synthesis Using Esterel. In *Proc of the International Workshop on Logic Synthesis*, May 1993.

[2] C. Berthet, O. Coudert, and J.C. Madre. New Ideas on Symbolic Manipulations of Finite State Machines. In *Proc of ICCAD*, October 1990.

[3] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A Structural Approach to State Space Decomposition for Approximate Reachability Analysis. In *Proc of the ICCD*, October 1994.

[4] O. Coudert, J.-C. Madre, and H. Touati, December 1993. TiGeR Version 1.0 User Guide, Digital Paris Research Lab.

[5] G. DeMicheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State Assignment for Finite State Machines. *IEEE Trans. on CAD*, 4(3):269–285, July 1985.

[6] J. Hartmanis. On the State Assignment Problem for Sequential Machines. *IRE Trans on Electronic Computers*, EC-10(2):157–165, June 1961.

[7] B. Lin and A. Richard Newton. Exact Redundant State Registers Removal Based on Binary Decision Diagrams. In *Proc of the International Workshop on Logic Synthesis*, mai 1991.

[8] S. Malik, E.M. Sentovich, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques. *IEEE Trans on CAD*, CAD-10(1):74–84, January 1991.

[9] S. Quer, G. Cabodi, P. Camurati, L. Lavagno, E.M. Sentovich, and R.K. Brayton. Incremental FSM Re-Encoding for Simplifying Verification by Symbolic Traversal. In *Proc of the International Workshop on Logic Synthesis*, May 1995.

[10] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc of the ICCD*, pages 328–333, October 1992.

[11] T. Shiple, G. Berry, and H. Touati. Constructive Analysis of Cyclic Circuits. In *ED-TC*, pages 328–333, March 1996.

[12] C.A.J. van Eijk and J.A.G. Jess. Detection of Equivalent State Variables in Finite State Machine Verification. In *Proc of the International Workshop on Logic Synthesis*, May 1995. Appeared in the poster session.

[13] T. Villa and A.L. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. *IEEE Trans on CAD*, CAD-9(9):905–924, September 1990.