Integrating Logic Synthesis, Technology Mapping, and Retiming

Alan Mishchenko Satrajit Chatterjee Robert Brayton
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
{alanmi, satrajit, brayton}@eecs.berkeley.edu

Peichen Pan
Magma Design Automation
12100 Wilshire Blvd., Ste 480
Los Angeles, CA 90025
peichen@magma-da.com

ABSTRACT
This paper presents a synthesis method that combines logic synthesis, technology mapping, and retiming into a single integrated flow. The proposed integrated method is applicable to both standard cell and FPGA designs. An efficient implementation is proposed using sequential And-Inverter Graphs. Experiments on a variety of industrial circuits from the IWLS 2005 benchmark set show an average reduction of the clock period of 25%, compared to the traditional mapping without retiming, and by 20%, compared to traditional mapping followed by retiming applied as a post-processing step.

1 INTRODUCTION
In recent years, the development of logic synthesis algorithms has reached a point of convergence, leading to the integration of different aspects of the synthesis process. This tendency is motivated by the shrinking of DSM technologies, which forces more of the synthesis aspects to be considered as interrelated and computed simultaneously. Some recent examples of this convergence can be found in the research work on integrating:

1. Tech-independent synthesis and mapping [21][7][20]
2. Mapping and retiming [28][32][11][12]
3. Retiming and placement [3][9]
4. Re-synthesis and retiming [3][33]
5. Tech-independent synthesis and placement [6][19][16]
6. Re-wiring and placement [8]
7. Clock skewing and placement [17]

Integrated methods explore several solution spaces at once; a solution found by an exact algorithm for an integrated approach is always better than one where an optimum solution is found in one space and fixed before optimizing it in the next space, and so on. Generally, the same applies to heuristic algorithms as well.

To illustrate the use of integration, consider logic synthesis and technology mapping. If these steps are not integrated, the network is first optimized by technology-independent logic synthesis. The resulting network is then given to delay-optimal technology mapping resulting in the best delay for the *given* logic structure. However, the decisions made during tech-independent synthesis are independent from technology mapping and so logic structures leading to a good mapping are often lost.

In order to integrate logic synthesis and technology mapping, it was proposed [21][7] to collect logic structures seen during logic transformations in technology-independent logic synthesis and simultaneously subject all of them to mapping. As a result, the choice of the best logic structure is made during mapping, when more accurate timing information is available.

The contribution of this paper is three-fold:

(1) *Integration.* Although double integrations, such as synthesis/mapping and mapping/retiming, have been studied, this paper is the first to develop a triple integration, where the three optimization spaces (synthesis/mapping/retiming) are explored simultaneously. The approach, called *sequential integration,* is exact because it finds the minimum delay among all possible synthesized, mapped, and retimed networks. In fact, it finds the global delay optimum solution by a sequence of *simple local* steps.

(2) *Applicability.* In the literature, there has been work on integrating synthesis and mapping for standard cells [21] and on integrating mapping and retiming for FPGAs [32]. By unifying with the technology mapping step, we show that the triple integration is possible, and, with minor variations, is applicable to both standard cells and FPGAs.

(3) *Efficiency.* One difficulty of integrated methods is that the combined optimization space is much larger than any individual one, typically leading to an increase in runtime. The third contribution is in demonstrating that searching the combined space of synthesis/mapping/retiming can be implemented efficiently using And-Inverter Graphs (AIGs) and the related cost functions. The experiments confirm that our implementation for both standard cells and FPGAs is highly scalable, processing industrial circuits with 100K+ gates in about one minute on a typical computer.

In the presentation and our current implementation, we limit ourselves to designs with single clock domain and edge-triggered D-flip-flops (possibly with initial states). However, we shall point out that the framework can be extended to handle designs with multiple clock domains and explicit set/reset logic.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 presents the integration procedures. Section 4 discusses the role of AIGs for efficient implementation. Section 5 shows experimental results. Section 6 concludes the paper and outlines future work.

2 BACKGROUND
A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to the wires. AIG is a Boolean network composed of two-input ANDs and inverters. The terms network, Boolean network, design, and circuit are used interchangeably.

Each node has a unique integer number called the *node ID.* A node has zero or more *fanins,* i.e. nodes that are driving this node, and zero or more *fanouts*, i.e. nodes driven by this node. The *primary inputs* (PIs) of the network are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. Terms memory elements, flop-flops, and registers are used interchangeably in this paper.

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. A *trivial cut* of the node is the cut composed of the node itself. A cut is *K-feasible* if it has *K* leaves or less.

A transitive *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path length but the PIs are not.

For standard cells, we use a load independent delay model for our experiments since we target a gain-based flow where sizing and buffering are done after mapping in conjunction with physical synthesis.

The area and delay of an FPGA mapping is measured by the number of LUTs and the number of LUT levels respectively. The delay of a standard cell mapping is computed using pin-to-pin delays of gates assigned to implement a cut. The load-independent timing model is assumed throughout the paper.

3 INTEGRATION

We view technology mapping as the core procedure and show how retiming and combinational logic synthesis fit in with this. Section 3.1 summarizes the mapping procedure. Section 3.2 shows how to combine mapping and retiming. Section 3.3 completes the description by combining the above steps with combinational logic transformations.

3.1 Technology mapping

In this section, we briefly describe each step of the Boolean mapping procedure and refer the reader to [7][28][29] for details.

- **Preparing the circuit for mapping**

The preparation is done by deriving a balanced AIG. For this, the SOP representations of the node functions are factored [4]. The AND and OR gates of the factored forms are converted into two-input ANDs and inverters and added to the AIG manager while performing one level structural hashing [15]. The resulting AIG is balanced by applying the associative transform, $a(bc) = (ab)c$, to reduce the number of AIG levels. Balancing can be done optimally in a linear topological sweep from the inputs.

- **Computing *K*-feasible cuts**

The cut computation [32][13] starts at the PIs and proceeds in the topological order to the POs. For a PI, the set of cuts contains only the trivial cut. For an internal node n with two fanins, a and b, the cuts $\Phi(n)$ are computed by *merging* the cuts of a and b:

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(a), u \in \Phi(b), |u \cup v| \le k\}.$$

Informally, merging two sets of cuts adds the trivial cut of the node to the set of pair-wise unions of cuts belonging to the fanins, while keeping only K-feasible cuts. A modification for networks with structural choices is discussed in Section 3.3.2.

In addition, sequential cuts, extending over the register boundary, can be computed and used to improve the quality of mapping for the sequential circuits. This is used in our integrated flow. We omit the details of this iterative computation of the sequential cuts due to page limitations, and refer the reader to [32].

- **Computing Boolean functions of cuts**

This step is needed only for standard cell mapping. For each cut, the root node's function is found in terms of the cut variables. In FPGA mapping, we only need to do this once for the best cuts, which are used to produce the final netlist after mapping.

- **Matching cuts with LUTs or gates**

In FPGA mapping, matching is performed by associating K-feasible cuts with K-input LUTs, which will implement it. For standard-cells, Boolean matching is more complex and involves associating each cut with a gate (or sets of gates), which can implement the Boolean function of the cut, allowing possible permutation and complementation of the inputs of the gate.

- **Assigning delay-optimal matches at each node**

In a topological order from the PIs, the match having the smallest delay is assigned to each node. This is the basic DAG-mapping step, which leads to the optimum-delay matching for both FPGAs [10] and standard cells [22]. When standard cells are used, mapping both phases of the nodes helps reduce the delay when making a phase assignment of the fanins of the gates [7].

- **Recovering area using heuristics**

The best area match (that preserves the minimum delay) at each node of the network is updated several times, resulting in a new mapping with smaller area [28][29].

- **Choosing the final mapping**

This is done in reverse topological order; first, we start the mapping with those LUTs or gates that are needed to implement the best cuts of the POs. Then, we visit the leaves of these cuts and add their implementation to the mapping, continuing recursively to the PIs. The procedure to derive the final mapping for sequential circuits is the same.

3.2 Combining mapping with retiming

Mapping for standard cells and FPGAs can be extended to sequential circuits by considering registers as labels (or weights) on the edges connecting logic nodes; the DAG becomes a cyclic circuit with labels. The overall mapping procedure for cyclic circuits (called *sequential mapping*) is similar to the traditional combinational mapping with a few modifications: (1) the concept of arrival times is extended to account for register labels on the edges; (2) computation of the arrival times is done by iterating over the circuit, and; (3) the resulting mapping has a retiming associated with it, which when performed on the mapped circuit, leads to the minimum clock period over all possible mappings and retimings. Below, we describe these modifications in detail.

Similar to the case for combinational mapping, computing and matching the cuts is done only once, at the beginning of mapping. However, the computation of sequential arrival times may be repeated for different clock periods as well as during area recovery.

3.2.1 Sequential arrival times

The sequential delay of a (possibly cyclic) path p is computed as: $l(p) = \sum_{n \in p} d(n) - \phi \sum_{e \in p} t(e)$, where $d(n)$ represents the delay for a node n and $t(e)$ represents the number of registers on the edge e. Thus the sequential delay is the difference between the sum of delays of nodes on the path and the clock period, ϕ, times the total number of registers on the path. The reason is that each register delays the signal at the end of the path by one clock cycle. Similar to the combinational case, the sequential arrival time (or l-value in the terminology of [32]) at node n is the maximum of the arrival times of all (possibly cyclic) paths originating at a PI and ending at node n: $l(n) = \max_{p \in PATH(PI \to n)} l(p)$.

As in the combinational case, the clock period ϕ is infeasible if the arrival time at a PO exceeds ϕ at any time during the iterative computation.

3.2.2 Iterative computation of sequential arrival times

These are computed for all nodes in the circuit G iteratively, as shown in Figure 1 [31][32]. The arrival times of the PI nodes are set to 0 and those of the internal nodes and the POs are initialized to -∞. In each iteration, nodes are visited in some order and their new arrival times are computed as: $l_{new}(n) = \min_{M \in \text{matches}(n)} \max_{u \in \text{fanin}(M)} \{l(u) - t_{u \to n}\phi + d_{u \to n}\}$

where $t_{u \to n}$ is the number of registers on the fanin edge and $d_{u \to n}$ is the pin-to-pin delay of a match for that node. Thus, for each node, we consider all possible matches and choose the one which yields the smallest new arrival time. Since the sequential cuts are pre-computed at the begining of mapping and stored, this computation is fast.

The arrival time of the node is updated if the new value is larger than the old value. Thus the arrival time at a node increases monotonically during the computation. If ever the arrival time at any PO exceeds ϕ, the iteration is stopped and the clock period is declared infeasible. Otherwise, the arrival times will converge and the clock period is feasible.

status **SequentialArrivalTimes** (circuit G, clock period ϕ)
for each node n in G **do**
if n is a PI **then** $l(n) = 0$ **else** $l(n) = -\infty$
do {
for each non-PI node n in G **do**

$$l_{new}(n) = \min_{M \in \text{matches}(n)} \max_{u \in \text{fanin}(M)} \{l(u) - t_{u \to n}\phi + d_{u \to n}\}$$

$$l(n) = \max\{l(n), l_{new}(n)\}$$

if n is a PO and $l(n) > \phi$
return INFEASIBLE
} **while** (the arrival times of some nodes have changed)
return FEASIBLE

Figure 1. Iterative computation of sequential arrival times.

To find an optimum clock period, a binary search is performed starting from a lower bound (set to 0) and an upper bound (set to the delay of the longest combinational path). In each step of the binary search, the iterative procedure in Figure 1 is repeated until a convergence criterion is met. We emphasize that when the sequential arrival times are computed, in essence, mapping and synthesis are done at the same time since we have all cuts and all matches (obtained using choice nodes) pre-computed, and we find the overall best match according to the sequential arrival time computation.

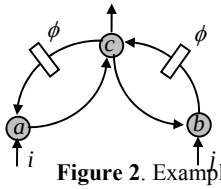
3.2.3 Retiming associated with the final mapping

When the optimum clock period, ϕ^{opt}, is known, a mapping is selected as described in Section 3.1. For each node n included in the mapping, having sequential arrival time $l^{opt}(n)$ computed for ϕ^{opt}, the final retiming is computed using the formula [32]:

$$r(n) = \begin{cases} 0, \text{ if } n \text{ is a PI or PO} \\ \left\lceil \frac{l^{opt}(n)}{\phi^{opt}} \right\rceil - 1, \text{ otherwise} \end{cases}$$

If the mapped circuit is retimed using this formula, the resulting clock period can be slower than the optimum clock cycle ϕ^{opt} by at most the delay of a gate [34]. When the unit delay model is used in the fixed-LUT-size FPGA mapping, this retiming gives the optimum clock period.

Example. The network in Figure 2 illustrates the computation of the sequential arrival times for the clock periods of 1 and 2 shown in the tables. The combinational delay of an internal node is 1. The longest combinational path (*a, c, b*) has delay 3. Initially, the arrival times of the PIs, *i* and *j*, are set to 0, and arrival times of the internal nodes, *a*, *b*, and *c*, are set to -∞. The clock period of 1 is infeasible because the arrival times of the PO node *c* exceeds the clock period after the first iteration. The clock period of 2 is feasible because the arrival times converge after two iterations. The associated retiming is $r(a) = r(c) = 0$, $r(b) = 1$. Indeed, if the register on the edge (*b, c*) is retimed backward over node *b*, the longest combinational path has delay 2.



$\phi = 1$

Iter	*a*	*b*	*c*
0	-∞	-∞	-∞
1	1	1	2
2	2	3	3
3	3	4	4

$\phi = 2$

Iter	*a*	*b*	*c*
0	-∞	-∞	-∞
1	1	1	2
2	1	3	2
3	1	3	2

Figure 2. Example of sequential arrival time computation.

3.3 Combining mapping with synthesis

The search space of combinational logic synthesis is added to those of mapping and retiming, by deriving and storing multiple circuit structures, produced during logic optimization, instead of considering only one circuit produced at the end.

Accumulating different logic structures implementing the same function is useful for two reasons. First, technology-independent synthesis is heuristic and produces a network that is not optimal. When this network is mapped, the mapper may fail to find a good set of matches, which might exist for an intermediate network in the flow. Second, synthesis operations usually apply to the network as a whole. Optimization for delay may significantly increase area, since the whole network, and not just the critical path, is optimized for delay. By combining a delay-optimized network with an area-optimized one, the mapper gets the best of both; on the critical path, logic structures from the delay-optimized network will be used, whereas off the critical path, the mapper may choose structures from the area-optimized network. Similarly, multiple logic structures give additional freedom that can be exploited by retiming to find a shorter clock period.

The accumulation of choices has two effects on our overall flow. First it increases the number of matches at each node which can improve the sequential arrival times. Second, when area is recovered at the end, there are more options for this because there are more matches at each node.

The next two subsections discuss efficient construction of the choice network and extended cut computation to handle choices.

3.3.1 Constructing the choice network

The choice network is constructed from a collection of functionally equivalent networks. Recent advances in equivalence checking are used to identify functionally-equivalent, structurally-different internal points in the networks [21][24].

The choice network is an AIG derived by combining the original functionally equivalent networks, while collecting the internal nodes in the equivalence classes according to their global function. We use random simulation to identify potentially equivalent nodes, and then use a SAT engine to verify equivalence and construct the equivalence classes. To this end, we implemented a package called FRAIG (Functionally Reduced And-Inverter Graphs) that exposes the APIs comparable to those of a BDD package but internally uses simulation and SAT [27].

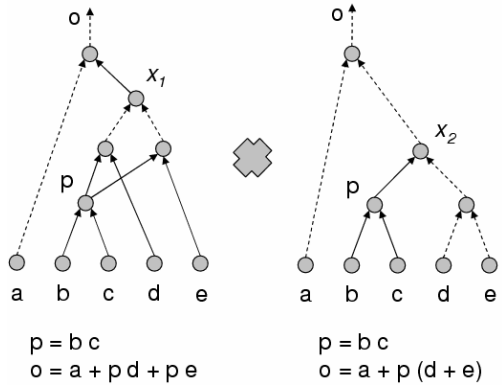


Figure 3. Equivalent networks before choicing.

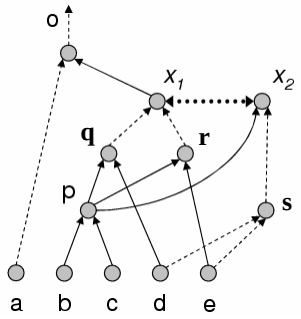


Figure 4. The choice network.

Example. Figures 3 and 4 illustrate construction of a network with choices. Networks 1 and 2 in Figure 3 show the subject graphs obtained from two networks that are functionally equivalent but structurally different. The nodes x_1 and x_2 in the two subject graphs are functionally equivalent (up to complementation). They are combined in an equivalence class in the choice network, and an arbitrary member (x_1 in this case) is set as the class representative. Node p does not lead to a choice because p is structurally the same in both networks. There is no choice corresponding to the output node o since the procedure detects the maximal commonality between the two networks.

A different way of generating choices is by iteratively applying the Λ- and Δ-transformations [23]. Given an AIG, we use the associativity of the AND operation to locally re-write the graph (the Λ-transformation), i.e. whenever the structure $(x_1x_2)x_3$ is seen in the AIG, it is replaced by the equivalent structures $(x_2x_3)x_1$ and $(x_1x_3)x_2$. If this process is done until no new AND nodes are created, it is equivalent to identifying the maximal multi-input AND-gates in the AIG and adding all possible tree decompositions of these gates. Similarly, the distributivity of AND over OR provides another source of choices.

Using structural choices leads to a new way of thinking about logic synthesis: rather than trying to come up with a good final netlist used as an input to mapping, we accumulate choices by applying a sequence of transformations, each of which leads to improvement in some sense. The best combination of these choices is selected during mapping, retiming, or combined mapping/retiming, leading to the triple integration of synthesis/mapping/retiming.

3.3.2 Cut enumeration with choices

The cut-based structural FPGA mapping procedure can be extended naturally to handle equivalence classes of nodes. Given a node n, let N denote its equivalence class. Let $\Phi(N)$ denote the set of cuts of the equivalence class N. Then, $\Phi(N) = \bigcup_{n \in N} \Phi(n)$, where, if a and b are the two inputs of n belonging to equivalence classes A and B, respectively, then

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(A), u \in \Phi(B), |u \cup v| \le k\}.$$

This expression for $\Phi(n)$ is a slight modification of the one used in Section 3 to compute the cuts without choices. The cuts of n are obtained from the cuts of the *equivalence classes* of its fanins (instead of the cuts of its fanins). In the absence of choices (which corresponds to the situation when each equivalence class has only one node) this computation is the same as the one presented in Section 3. As before, the cut enumeration is done in one topological pass from the PIs to the POs in combinational designs and multiple passes in sequential designs.

Example. Consider the computation of the 3-feasible cuts of the equivalence class $\{o\}$ in Figure 4. Let X represent the equivalence class $\{x_1, x_2\}$. Now, $\Phi(X) = \Phi(x_1) \cup \Phi(x_2) = \{\{x_1\}, \{x_2\}, \{q, r\}, \{p, s\}, \{q, p, e\}, \{p, d, r\}, \{p, d, e\}, \{b, c, s\}\}$. We have $\Phi(\{o\}) = \Phi(o) = \{\{o\}\} \cup \{u \cup v \mid u \in \Phi(\{a\}), u \in \Phi(\{x_1\}), |u \cup v| \le 3\}$.

Since $\Phi(\{a\}) = \Phi(a) = \{a\}$ and $\Phi(\{x_1\}) = \Phi(X)$, we get $\Phi(\{o\}) = \{\{o\}, \{a, x_1\}, \{a, x_2\}, \{a, q, r\}, \{a, p, s\}\}$. Observe that the set of cuts of o involves nodes from the two choices x_1 and x_2, i.e. o may be implemented using either of the two structures.

The subsequent steps of the mapping process (computing delay-optimum mapping and performing area recovery) remain unchanged, except that mapping at each node is done using the additional cuts. These cuts as well as the additional cuts overlapping with the gate boundary (sequential cuts) represent orthogonal ways to improve the quality of mapping.

3.4 Overall picture of sequential integration

Figure 5 illustrates the overall flow, emphasizing where the various computations enter the picture. Computation begins by accumulating functionally equivalent networks, which are processed by the FRAIG manager, resulting in the choice network. Next, the cuts are computed for each node, and matches are found for each cut. Then a binary search is performed to find the best achievable clock period. If the result is *not okay*, additional synthesis can be applied, resulting in more and better choices, which may further improve the clock period. Once the target clock period is found, the associated retiming is performed, and the final network, after area recovery, is produced. Note that steps denoted *cuts and matches* and *seq arrival times* involve iteration over the sequential AIG until convergence.

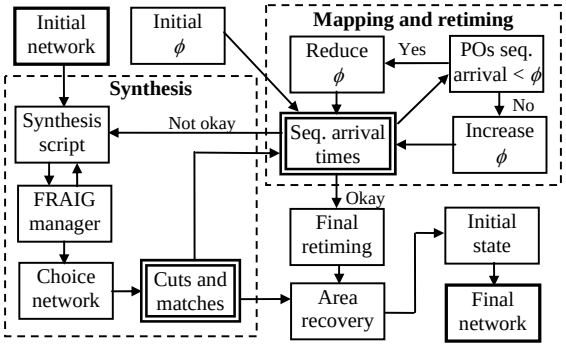


Figure 5. High-level view of the integration flow.

4 IMPLEMENTATION DETAILS

AIGs have been used successfully in a number of logic synthesis and verification projects [21][34]. To implement the integrated flow, the combinational AIGs are generalized to handle sequential transformations as shown in [1]. In the resulting sequential AIGs, each edge has four attributes: (1) a node ID; (2) a complemented attribute; (3) the number of registers; and (4) the vector of initial states of the registers. Using sequential AIGs leads to an efficient implementation for the following reasons:

- During the cut computation only two cut sets are merged at each two-input AND node. Storing register numbers together with node IDs on the edges allows for a convenient manipulation (e.g. hashing) of the leaves of sequential cuts.
- The iterative arrival time computation is fast using the AIG because this representation is uniform and compact.
- Retiming of a sequential AIG is simple because the register numbers and initial values are stored on the graph edges.
- Computation of the initial state in backward retiming is reduced to a SAT problem, which "records" the sequence of backward register movements during retiming. (Computation of initial state for forward retiming is easy.) Retiming of networks with arbitrary gates and logic nodes is reduced to retiming of a sequential AIG constructed to reflect the structure of the given network.

5 EXPERIMENTAL RESULTS

The integrated flow was implemented in the sequential logic synthesis and verification system ABC [2]. The ABC commands *smap* and *sfpga* perform the integrated optimization for standard cells and FGPAs, respectively. The implementation was tested on the designs included in the IWLS `05 benchmark set [18]. Table 1 shows the results of FPGA mapping into 5-input LUTs. Table 2 shows the results of standard-cell mapping using the library, *mcnc.genlib,* from the standard distribution of SIS. The netlists after mapping were verified using a bounded SAT-based sequential equivalence checker in ABC.

The following notation is used in the tables. The first column lists the benchmarks. The next five columns show the number of primary inputs (PI), primary outputs (PO), registers (Register), AIG nodes (AND2), and maximum number of logic levels of the AIG (Lev). The

number of gates and logic levels is given for an AIG after structural hashing and algebraic balancing for minimum delay, as described in Section 3.1. The next five columns show the clock periods after optimization with different options:

- M – combinational mapping only
- MC – combinational mapping with structural choices (integrated synthesis/mapping)
- M+R – combinational mapping followed by retiming
- MR – sequential mapping (integrated mapping/retiming)
- MC+R – integrated synthesis/mapping followed by retiming
- MCR – sequential mapping with structural choices proposed in this paper (integrated synthesis/mapping/retiming)

The last two columns in the tables show the runtime, in seconds, of the two most time-consuming steps in the fully integrated computation. Column "Cut" gives the runtime of the exhaustive computation of 5-feasible sequential cuts. Column "Iter" gives the total runtime of the binary search for the optimum clock period. This runtime is dominated by iterative computation of the sequential arrival times. The runtimes are reported on a 1.6GHz laptop. The memory requirements, not listed in the tables, were dominated by the sequential cuts and were less than 500MB for the largest circuits considered.

We also experimented with the ISCAS benchmarks and obtained very close agreement with the results reported in Table 2 on the average improvements in the clock period for both standard cells and FPGAs. The details for individual benchmarks are omitted due to the page limitation.

To summarize, the experiments show that the proposed approach often substantially reduces the optimum clock period over separate or partially-integrated approaches. The improvements achieved by the integrated optimizations (MR and MCR) are substantially better than those found by the consecutive application of the individual optimizations. The runtimes confirm the scalability of the proposed integrated flow.

6 CONCLUSIONS AND FUTURE WORK

This paper presents an approach synergistically integrating combinational logic synthesis, technology mapping, and retiming. The clock period found is provably the smallest one in the combined solution space of the above transformations. It gives good results because the multi-dimensional search space contains deeper minima than the projection of this space on any specific dimension. The approach is highly scalable because global minimization is achieved by a sequence of simple local transformations. The proposed implementation, based on sequential AIGs, scales to large circuits and results in an average reduction of about 25% in clock period for both standard cells and FPGAs.

Future work will focus on the following improvements:

- *Register minimization after retiming.* Our current proof-of-concept implementation does not attempt to produce the final retiming with the minimum number of registers. Besides sharing registers across fanins and fanouts, the number of registers can be reduced by either an exact linear program [31], or simpler heuristics that move registers to better positions using local sequential slacks.
- *Area recovery for sequential circuits.* Area recovery for combinational circuits proceeds from inputs to outputs. The required times do not change and, therefore, need not be recomputed. However, for a cyclic circuit, both sequential arrival and required times have to be recomputed whenever a node's timing is changed during area recovering. An efficient method for updating this information is required.
- *Convergence speed of iterative procedures*. Sequential arrival time computation in Figure 2 often iterates over the circuit many times before converging or proving a clock period infeasible. Also, the arrival times are recomputed from scratch whenever a new clock period is assumed. These inefficiencies may be addressed using Howard's algorithm [14] to detect the critical cycles and avoid re-computing the timing information for the non-critical nodes.
- *Generation of structural choices for sequential networks*. Our current procedures only generate choices from the un-retimed version of the sequential circuit. We consider extending choice generation to sequential networks by combining the combinational choices derived for the original network and networks with shifted register boundaries [26].

7 REFERENCES

[1] J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures", *Proc. ICCAD'01*, pp. 176-182.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. December 2005 Release. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] S. Bommu, N. O'Neill, and M. Ciesielski. "Retiming-based factorization for sequential logic optimization", *ACM TODAES*, Vol. 5(3), July 2000, pp. 373-398.

[4] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[5] T. F. Chan, J. Cong, T. Kong, and J. R. Shinnerl, "Multilevel optimization for large-scale circuit placement". *Proc. ICCAD '00*, pp. 171-176.

[6] S. Chatterjee and R. Brayton, "A new incremental placement algorithm and its application to congestion-aware divisor extraction", *Proc. ICCAD '04*, pp. 541-548.

[7] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05,* pp. 519-526.

[8] P. Chong, Y. Jiang, S. Khatri, F. Mo, S. Sinha, and R. Brayton, "Don't care wires in logical/physical design", *Proc.IWLS'00*, pp.1-9.

[9] P. Chong and R. Brayton, "Characterization of feasible retimings", *Proc. IWLS '01*, pp. 1-6.

[10] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", *IEEE Trans. CAD*, vol. 13(1), January 1994, pp. 1-12.

[11] J. Cong and C. Wu, "An efficient algorithm for performance-optimal FPGA technology mapping with retiming", *IEEE Trans. CAD*, vol. 17(9), Sep. 1998, pp. 738-748.

[12] J. Cong and C. Wu, "Optimal FPGA mapping and retiming with efficient initial state computation", *IEEE Trans. CAD*, vol. 18(11), Nov. 1999, pp. 1595-1607.

[13] J. Cong, C. Wu and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA `99*, pp. 29-35.

[14] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms", *ACM TODAES*, Oct. 2004, Vol. 9(4), pp. 385-418.

[15] M. K. Ganai, A. Kuehlmann, "On-the-fly compression of logical circuits", *Proc. IWLS '00*.

[16] W. Gosti, S. Khatri and A. Sangiovanni-Vincentelli. "Addressing the timing closure problem by integrating logic optimization and placement", *Proc. ICCAD'01*, pp. 224-231.

[17] A. P. Hurst, P. Chong, A. Kuehlmann, "Physical placement driven by sequential timing analysis". *Proc. ICCAD '04*, pp. 379-386.

[18] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[19] Y. Jiang and S. Sapatnekar. "An integrated algorithm for combined placement and libraryless technology mapping," *Proc. ICCAD '99*, pp. 102-106.

[20] V. N. Kravets. *Constructive multi-level synthesis by way of functional properties*. Ph.D. Thesis, University of Michigan, 2001.

[21] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *IEEE TCAD*, Vol. 21(12), Dec 2002, pp. 1377-1394.

[22] E. Y. Kukimoto, R. Brayton, P. Sawkar, "Delay-optimal technology mapping by DAG covering", *Proc. DAC '98*, pp. 348-351.

[23] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. CAD*, Vol. 16(8), 1997, pp. 813-833.

[24] F. Lu, L. Wang, K. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," *Proc. DAC `03*, pp. 668-673.

[25] N. Maheshwari and S. Sapatnekar. "Efficient retiming of large circuits", *IEEE Trans VLSI*, Vol. 6(1), March 1998, pp. 74-83.

[26] S. Malik, K.J. Singh, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis", *IEEE Trans. CAD*, Vol. 12(5), May 1993, pp. 568-578.

[27] A. Mishchenko, S.Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report*, EECS Dept., U. C. Berkeley, March 2005.

[28] A. Mishchenko, S. Chatterjee, R. Brayton, and M. Ciesielski, "An integrated technology mapping environment", *Proc. IWLS '05*, pp. 383-390.

[29] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs", *Proc. FPGA '06 (to appear).*

[30] P. Pan and C. L. Liu, "Optimum clock period FPGA technology mapping for sequential circuits", *Proc. DAC '96*, pp. 720-725.

[31] P. Pan, "Continuous retiming: Algorithms and applications. *Proc. ICCD '97*, pp. 116-121.

[32] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs", *Proc. FPGA '98*, pp. 35-42.

[33] P. Pan, "Performance-driven integration of retiming and resynthesis", *Proc. DAC '99*, pp. 243-246.

[34] M. Papaefthymiou, "Understanding retiming through maximum average-delay cycles", *Math. Syst. Theory*, No. 27, 1994, pp. 65-84.

[35] J. S. Zhang et al, "Simulation and satisfiability in logic synthesis", *Proc. IWLS '05*, pp. 161-168.

Table 1. Integration of synthesis/mapping/retiming for FPGAs (k = 5).

IWLS 2005 benchmarks	Network statistics					Optimal clock period						Runtime	
	PI	PO	Register	AND2	Lev	M	MC	M+R	MR	MC+R	MCR	Cut, s	Iter, s
ac97_ctrl	84	48	2199	14261	11	3	3	3	2	3	2	2.56	1.69
aes_core	259	129	530	21125	21	6	6	6	6	6	6	4.15	5.61
des_area	240	64	128	4781	27	8	8	8	8	8	8	0.57	0.42
des_perf	234	64	8808	76716	17	5	5	5	4	5	3	16.97	9.32
ethernet	98	115	2235	19654	27	8	8	8	7	8	5	4.76	5.21
i2c	19	14	128	1151	12	4	4	4	4	4	3	0.15	0.12
mem_ctrl	115	152	1083	15191	28	9	8	9	8	8	6	1.32	1.85
pci_bridge32	162	207	3359	22742	22	7	7	7	6	7	6	5.92	4.52
pci_spoci_ctrl	25	13	60	1369	16	5	5	5	5	5	4	0.27	0.31
sasc	16	12	117	772	8	3	2	3	2	2	2	0.12	0.05
simple_spi	16	12	132	1031	10	3	3	3	3	3	3	0.22	0.15
spi	47	45	229	3768	31	8	8	8	6	8	6	0.60	0.71
ss_pcm	19	9	87	405	7	2	2	2	2	2	2	0.04	0.03
systemcaes	260	129	670	12279	44	10	9	8	7	9	6	2.88	3.42
systemcdes	132	65	190	2933	23	7	7	5	4	6	4	0.36	0.31
tv80	14	32	359	9503	42	14	12	12	9	11	8	1.24	2.62
usb_funct	128	121	1746	15670	23	7	7	7	6	6	5	2.45	3.96
usb_phy	15	18	98	456	9	3	3	3	2	3	2	0.07	0.04
vga_lcd	89	109	17079	126687	19	7	7	7	5	7	4	23.01	21.00
wb_conmax	1130	1416	770	47535	18	7	7	7	5	7	5	3.53	2.32
wb_dma	217	215	563	4044	22	8	8	8	6	8	6	0.46	0.40
Ratio						**1.00**	**0.96**	**0.97**	**0.82**	**0.95**	**0.74**		

Table 2. Integration of synthesis/mapping/retiming for standard cells (*mcnc.genlib*)

IWLS 2005 benchmarks	Network statistics					Optimum clock period						Runtime	
	PI	PO	Register	AND2	Lev	M	MC	M+R	MR	MC+R	MCR	Cut, s	Iter, s
ac97_ctrl	84	48	2199	14261	11	8.30	8.00	8.00	6.13	7.50	5.69	3.48	9.29
aes_core	259	129	530	21125	21	17.30	17.00	17.76	17.19	17.22	16.80	5.81	24.67
des_area	240	64	128	4781	27	22.00	22.00	22.13	23.25	22.13	22.77	0.80	1.02
des_perf	234	64	8808	76716	17	14.80	13.60	14.80	11.81	12.19	10.83	22.90	30.25
ethernet	98	115	2235	19654	27	21.30	19.70	20.39	20.34	19.33	14.06	5.87	20.59
i2c	19	14	128	1151	12	10.00	10.00	10.13	8.00	10.13	7.50	0.21	0.76
mem_ctrl	115	152	1083	15191	28	21.10	21.90	21.30	18.89	21.66	15.02	1.83	6.81
pci_bridge32	162	207	3359	22742	22	17.70	17.10	16.31	15.03	16.12	15.23	7.46	17.56
pci_spoci_ctrl	25	13	60	1369	16	12.10	11.00	12.34	10.94	10.97	8.91	0.35	1.48
sasc	16	12	117	772	8	7.40	6.80	7.05	6.91	6.32	6.09	0.16	0.42
simple_spi	16	12	132	1031	10	9.10	9.10	9.02	7.50	8.67	7.03	0.30	0.95
spi	47	45	229	3768	31	21.80	19.70	17.85	17.00	15.60	16.00	0.80	3.12
ss_pcm	19	9	87	405	7	6.60	6.30	5.91	4.50	5.71	4.13	0.07	0.23
systemcaes	260	129	670	12279	44	28.10	25.20	26.34	24.57	24.65	22.78	3.99	12.85
systemcdes	132	65	190	2933	23	19.90	19.20	15.81	13.56	15.24	13.56	0.50	2.06
tv80	14	32	359	9503	42	33.90	31.80	31.13	26.72	29.58	24.96	1.74	11.65
usb_funct	128	121	1746	15670	23	19.70	17.20	19.33	14.00	17.40	13.41	3.24	13.47
usb_phy	15	18	98	456	9	7.10	6.80	7.11	6.50	6.87	5.25	0.10	0.35
vga_lcd	89	109	17079	126687	19	15.50	14.60	15.59	11.63	14.53	9.52	30.17	126.76
wb_conmax	1130	1416	770	47535	18	15.90	15.90	15.10	12.58	14.82	11.86	4.87	11.77
wb_dma	217	215	563	4044	22	18.60	17.80	17.70	15.61	16.71	14.22	0.56	2.36
Ratio						**1.00**	**0.95**	**0.96**	**0.84**	**0.91**	**0.76**		