Layout-driven Logic Optimization

R. Carragher, R. Murgai — Fujitsu Labs of America
S. Chakraborty* — IIT Mumbai
M. R. Prasad* — UC Berkeley
A. Srivastava* — Northwestern U.
N. Vemuri* — U. of Mass.

Abstract

With the advent of deep sub-micron technologies, interconnect loads and delays are becoming dominant. Consequently, the currently used design flow of iteratively performing logic synthesis with statistical wire-load models, doing placement & routing, extracting parasitics, and using them back in the synthesis tool runs into serious timing convergence problems. Layout-driven synthesis has become the need of the day. In this paper, we present a layout-driven optimization methodology that incorporates logic-level transformations for improving the delay and area of a mapped, block-placed, and globally routed design under certain constraints such as area availability, congestion, hold-time, and pin drive, etc. We argue that this is the right stage for optimization. Only those transformations that are local, incremental, and layout-friendly are incorporated. Examples of such transformations are net buffering, gate resizing, generalized DeMorgan transform, and pin permutation. We propose a simple flow to simultaneously improve the delay and area of the design under these constraints. We applied this flow on large, already optimized, industrial designs and obtained significant delay and area improvements.

1 Introduction

Meeting performance goals is one of the most important design tasks. It can be addressed at various levels during the design flow: behavioral, architectural, gate, or layout. For today's deep sub-micron technologies, wire delays are becoming dominant. Therefore it becomes crucial to consider wire delays while minimizing delays at the gate-level. For instance, in today's design flow, a designer or a synthesis tool constructs buffer trees during logic optimization/mapping to improve the delay characteristics of the circuit as well as to redistribute the loads driven by the signals. However, since wire delays and capacitances are not known, the timing and load information is inherently inaccurate. This inaccuracy is increasing with the advent of deep sub-micron technologies, where interconnect loads and delays form dominant components of the total load and delay. It is possible that faster and/or smaller trees can be constructed by considering more accurate load and delay information available during placement and routing.

We propose a layout-driven optimization methodology for improving the delay and area of a design under certain constraints imposed by the layout and the cell library. We have implemented this methodology as a tool which works closely with a placement and routing tool. Our tool provides delay information to the layout tool, which uses it to refine the layout to further improve the delay.

The rest of the paper is organized as follows. In Section 2 we review previous work addressing the timing convergence problem. Section 3 discusses our optimization paradigm, which consists of a model and an optimization methodology. In Section 4, we briefly describe the constraints, optimization modes, and the transforms used in our implementation of the proposed methodology. Section 5 presents experimental results validating the efficacy of our approach and we conclude with some directions for future work in Section 6.

*This work was done when these authors were at Fujitsu Labs of America either as researchers or summer interns.

2 Related Work

Recently, there have been several works that address the issue of timing convergence in high performance design. They attempt to either supplement or replace the traditional approach of iterating between synthesis and physical design, by augmenting synthesis runs with physical data from the previous iteration. These can be broadly classified into the following three categories, *pre-layout estimation*, *unification based*, and *pre-routing optimization*.

The pre-layout estimation approach attempts to estimate the effects of layout without actually performing it, and then using those estimates to guide the optimization at a pre-layout stage. Abouzeid *et al.* [1] introduced lexicographical factorization, which is used to extract sub-functions from a function such that the resulting layout is more regular and ordered. In [18], Pedram and Bhat described a method in which a fast global placement is applied to the logic network (decomposed into two-input gates) and is used to guide the wiring estimation as well as technology mapping. Later, they extended this approach to include logic restructuring and technology decomposition [17]. This was further extended in [25] to include post-mapping fanout optimization. Here, an alphabetic-tree based fanout optimization technique was proposed to generate fanout trees free of internal edge crossings. One basic problem with this class of methods is that they *do not actually work on a real layout.* They predict or generate an intermediate layout and generate a netlist accordingly. The actual layout generated by the layout tool may be entirely different, resulting in the same non-convergence problem.

The unification based approach tries to perform a group of design and optimization steps simultaneously rather than sequentially. For example, [11, 5] combined technology mapping and placement; [20] combined floorplanning, technology mapping, and linear placement; [15, 21] combined routing tree construction and fanout optimization, and [10] combined logic restructuring and placement. Although these algorithms are promising in concept, they suffer from the following drawbacks. 1) They have been limited to very specific combinations of design steps and cost functions and have not been applied to the entire design flow. 2) They guarantee optimality on very restricted netlist topologies such as trees. The performance on general circuits remains unknown. 3) Since each individual design/optimization step is computationally hard, it is debatable if solving a combination of two or more such steps simultaneously is the right way to proceed. The applicability of such techniques to the complete design flow on large industrial designs and extensibility to arbitrary cost functions remains to be proven.

Finally, the pre-routing optimization approach tries to push certain optimization steps to later stages of the design flow where more authentic physical information is available. In the `LATTIS` system [3], a number of optimizations such as gate resizing, buffer insertion, timing directed factorization and remapping are performed on a technology mapped netlist (in contrast to a pre-mapped netlist). Kannan *et al.* [6] and Ishioka [4] proposed post-placement but pre-routing optimizations, followed by incremental placement to incorporate the changes. The set of transformations includes fanout optimization and gate resizing [6] and resynthesis & remapping [4]. In [6], the wire-routes are estimated by minimum spanning trees. In [23], Stenz *et al.* proposed a flow that combines netlist transformations with the detailed placement step. Hojat and Villarubia [19] described the results of using simple synthesis operations such as gate sizing between partitioning steps in a mincut based placement algorithm. More recently, Shenoy *et al.* [22] proposed a design flow that incorporates an intermediate iterative step between the traditional synthesis and physical design steps. This step integrates a number of synthesis optimizations in the inner loop of an iterative method to perform global placement under given timing constraints. The common drawback of all the pre-routing-based approaches is that the optimizations are performed before any sort of *true* routing information is available. Hence the assumptions regarding global net topology (and hence capacitive loading), which are crucial to fanout optimization and buffering, may be spurious.

Our methodology is similar in spirit to the last category of works, albeit with notable differences and improvements.

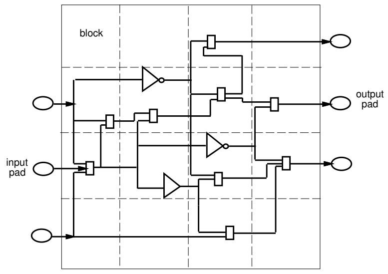

Figure 1: Model of our paradigm: block layout of a design

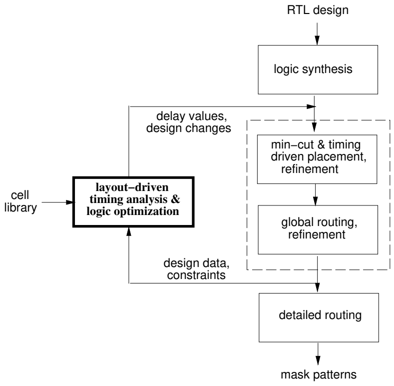

Figure 2: Proposed design flow

3 Our Paradigm

Our paradigm is based on the following diagnosis of the *timing convergence problem*, which is endorsed in part by several other researchers who have addressed this problem. The timing convergence problem can be attributed to the following:

- optimization steps that operate under incorrect assumptions or approximations of design data generated by other steps. e.g. synthesis steps using estimations of wire-loads.
- design and optimization steps that violate the assumptions made by other steps during their execution. e.g. physical design on a synthesis-optimized netlist produces a layout which violates the wire-load approximations used during synthesis.

While there is currently no consensus among the design community as to which approach (unification-based or pre-layout estimation-based or the current iteration-based or some combination thereof) is the *"true solution"* to the timing convergence problem, the following simple observation can certainly mitigate the problem to a large extent and still be practicable in an industrial setting. Consider a set of synthesis transformations applied to the design such that the following are true:

- Optimizations are applied at a stage in the design process where they operate on *real* design data rather than inexact approximations of it.
- Each optimization is applied in a *local* and *constraint-based* manner so as to cause only local perturbations in the global topology of the design.
- The optimizations are capable of yielding appreciable improvements in the quality of the final layout.
- Each optimization is algorithmically *simple* and *efficient* enough to be scalable yet *flexible* enough to handle a variety of cost functions which are commonplace in an industrial setting.

It can be argued that if one can realize these objectives, the said optimizations can produce appreciable gain in the quality of the design during a single pass of synthesis and physical design, thus directly aiding timing convergence. This observation forms the basis of our paradigm.

Our paradigm consists of a **model** and an **optimization methodology** which operates upon the model. The model, shown in Figure 1, is generated as follows. Given a logic-optimized, fully-mapped circuit, a min-cut based tool partitions the chip area into *blocks* using horizontal and vertical cuts. Depending on the total chip area and the number of blocks, the placement tool assigns each block a fixed total area, which constrains the number and types of gates that can be placed in that block. Each gate of the circuit is placed in a block (a block may contain more than one gate). The topology and global routing of each net is then determined by a global routing tool, which generates net segments between blocks (Figure 2). This fully-mapped, block-placed and globally routed design forms our model. The optimization methodology operates upon this model as follows. *Our methodology views designing a circuit to be a unified task with certain goals.* Examples of design goals are minimizing the circuit delay without violating the area and load constraints, fixing timing violations, recovering maximum area without increasing the circuit delay, fixing overloading problems, or a combination of these. Goals can be viewed as a combination of objectives and constraints. The constraints are imposed either by the specification of the design (e.g., design area, clock speed) or by the underlying technology (e.g., pin loading, hold time). We use a series of optimizing steps to achieve these objectives without violating any constraints. Each step applies a **transform** (such as gate resizing, net buffering, etc) to the design in a particular **optimization mode**. An optimization mode achieves an objective possibly subject to the given constraints. The transforms are carefully chosen so that they perturb the layout only slightly.

The design changes resulting from these transforms along with the timing information are passed to the layout tool, which further refines the placement and global routing by generating more cuts (Figure 2). The logic transformations may be invoked again on the refined design, with more accurate placement and wiring loads and delays. Thus, our paradigm embeds specific logic transforms *between successive refinement (or cut-generation) phases of a min-cut based placement tool and a global routing tool.* Note that this is different from the currently-used methodology of invoking logic resynthesis *after complete placement and routing*.

We believe that our paradigm captures the proper stage for applying optimization techniques such as buffering and gate-resizing. We have access to more physical details, but we are not overly constrained by the layout yet. After cells have been roughly placed and nets globally routed, wire loads and delays are known more accurately. The timing analysis performed on the design at this stage is quite accurate, since it uses delay parameters from the cell library, along with the *actual* wire loads and wiring delays. It is possible then that, for instance, buffer trees generated during logic optimization or mapping can be improved, resulting in better delay and area for the design. Also, since the layout is not final and detailed routing has not been done, any optimization gains obtained through the *careful addition and deletion of gates* can be incorporated during placement refinement and detailed routing that follow.

The proposed methodology is computationally efficient yet fairly flexible. It can use potentially any cost function and optimizations that can be made to operate in an incremental, local, and constraint-based setting. Also, since our methodology is orthogonal to most other proposed solutions for the timing convergence problem (viz. the pre-layout estimation approaches, unification based approaches or the current iteration based paradigm) it can be used to augment the optimizing power of any of these flows.

The key contributions of this paper can be summarized as follows:

- We propose a constraint-based, layout-driven logic optimization paradigm to combat the problem of timing convergence.
- As part of the proposed paradigm, we identify the stage in the design flow where the applied logic transformations can have a substantial yet predictable impact on the final quality of the design. See Figure 2.
- We identify a comprehensive set of synthesis transformations (namely gate sizing, buffer insertion/deletion, the generalized DeMorgan transform and pin permutation) and associated efficient algorithms, which are powerful enough to provide the requisite optimization yet local and incremental enough to be incorporated late in the design flow.
- We implemented this methodology in an industrial strength design tool that employs a varied set of cost functions such as timing, layout area, and pin loading and is designed to optimize designs with over half a million gates.
- We provide empirical evidence to demonstrate the efficacy of the proposed methodology. Specifically, the methodology is able to appreciably improve the area as well as delay of several *pre-optimized industrial designs using a reasonable amount of CPU time and memory.*

The following sections describe the implementation of the proposed methodology in the context of a particular industrial design tool and a discussion of experimental results demonstrating its efficacy.

4 Methodology Implementation

Based on discussions with the designers and our experience, we have implemented in our layout-driven optimization tool a set of constraints, optimization modes, and transforms. They work together to meet the design goals.

4.1 Constraints

Delay constraint: The delay of the circuit should not exceed D, a pre-specified value.

Area constraint: The total design area should not exceed A. Since in our paradigm the layout is divided into blocks, the global area constraint can be further refined into local constraints, one for each block. Each block P has a certain total area $T(P)$ assigned to it by the layout tool. The currently-used area of P plus the net area increase in P during optimizations (such as gate insertion, deletion, or resizing) must not exceed $T(P)$.

Load constraint: Each output pin of a library gate has a *drive strength*, which is the maximum capacitive load the pin can drive without compromising the integrity of the signal. Load constraint mandates that the maximum capacitive load driven by a pin should not be more than the drive strength of the pin.

Hold-time constraint: Given a memory element M (flip-flop, latch, etc.) in the design, the earliest arriving signal should arrive at the data input of M after the hold-time of M has elapsed.

4.2 Optimization Modes

Unconstrained delay minimization (UDM) The primary goal in this mode is to minimize the circuit delay. Minimizing the area penalty is only a secondary goal, i.e., when there is a tie between two choices that promise identical delay improvement, we pick the one with smaller area penalty. This mode is used when it is known that area and/or load constraints are not tight. Although unconstrained optimization is a special case of constrained optimization (set the constraint value to infinity), it may be better for efficiency purposes to have a separate implementation of the unconstrained optimizations. For instance, in the case of net buffering.

Constrained delay minimization (CDM) Here, the goal is to minimize the circuit delay under area, load, and hold-time constraints. Assuming an area-delay trade-off curve (which holds for most designs), the limited area resources on the chip make this the most important optimization mode.

Constrained area minimization (CAM) Area recovery is of utmost importance, because 1) It leads to reduction in the chip area and hence the chip cost. 2) Applying area recovery before delay optimization can free up a lot of area. Since most performance optimization techniques are area-hungry, this increases the flexibility and effectiveness of the performance optimization techniques. We will demonstrate this in Section 5. 3) Many performance optimization techniques are wasteful of area and rely on future runs of area recovery; for instance, Touati's global fanout optimization strategy [24]. Since our primary objective is delay improvement, area recovery should not reduce the minimum slack in the circuit.

Constrained overload fixing (COF) During layout, a pin that was satisfying its load constraint earlier can become overloaded. This mode has the following goal: *Given a design, determine the overloaded pins and fix them, incurring minimum delay & area penalties, without violating the area and hold-time constraints.*

4.3 Optimization Transforms

Now we briefly describe the logic transforms that optimize the circuit in various modes. The transforms are incremental and local; they perturb the logic structure and layout very slightly. To avoid timing convergence issues, the transforms work on the real layout and not a phantom one. The timing information is computed taking into account *actual* wire loads and delays, and the pin-to-pin cell delays using a real input slew-based delay model. Currently the set of transformations includes gate resizing, net buffering, the generalized DeMorgan transform, and pin permutation.

4.3.1 Gate Resizing

Gate resizing is a technique in which a gate is replaced by another gate with the equivalent logic function, but better area, drive strength, input loading, intrinsic delay, power consumption, or other features which improve the objective function. Because the replacement is in-place, there is very little effect on the placement and routing, and so this optimization can be performed at various stages of the design without requiring sophisticated engineering change tools. We have incorporated two modes in gate resizing: area-constrained delay minimization and delay-constrained area minimization. Unconstrained delay minimization is implemented as a special case of constrained delay minimization.

CDM It is known that the gate resizing problem for minimizing the circuit delay is NP-complete under various practical scenarios [8]. Therefore, one has to resort to heuristics. We have implemented several gate-resizing algorithms: greedy, dynamic programming-based, and sizing down non-critical fanouts of critical cells. They differ on how they pick the gates for resizing and the new sizes.

Greedy Algorithm The greedy algorithm first determines the best "local" size for each gate, g, in the critical subcircuit of the circuit C. Specifically, it recalculates delays in as many gates as necessary according to the delay model so that the slacks of all the pins of g can be correctly recalculated The size which provides the best slack is kept, as $s(g)$. Next, out of the entire set of best resizings for the circuit, only those that are useful (i.e., guarantee not to degrade

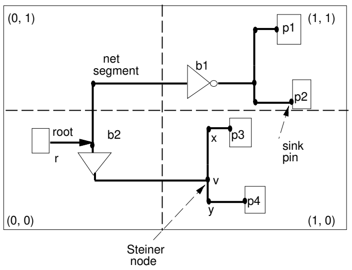

Figure 3: An extended net

the circuit's performance) are selected in the set S. A cost-benefit analysis is done to compute the set S. The circuit is then updated, i.e., the gate resizings contained in S are implemented, and the circuit's delays are recalculated. This process continues while there are useful gate resizings. When S is empty, the algorithm terminates. Note that when the set S of resizings is applied, the resizings are applied in order, from the "best" to the "worst." If during that application there is not enough area available for a resizing, then it is not performed. This preserves the area constraint.

Dynamic Programming-based Algorithm Kukimoto *et al.* [7] recently showed that a dynamic programming-based algorithm is optimum for minimum-delay technology mapping under the load-independent delay model and a library consisting of single-output gates. Although our circuits and libraries have multiple-output gates and the delay model is input slew-based (and not load-independent), we decided to experiment with the dynamic programming approach. First, we identify *regions* in the circuit. A region is a maximally connected sub-circuit consisting only of single-output gates. Then, we order the regions from inputs to outputs and apply the dynamic programming algorithm on each region in order, visiting the gates of the region topologically from region inputs to region outputs. After extensive experimentation, we found that this approach yields smaller delays on about half the designs as compared to the greedy one, although it is slower. Currently, we use both greedy and dynamic programming-based algorithms in our optimization script.

Size Down The previous two methods operate on critical gates of the circuit. One can also resize non-critical gates and obtain delay improvements. For instance, given a critical output pin p of a gate, the arrival time at p can be reduced if c_p, the capacitive load seen by p, can be reduced. For this, we identify non-critical fanout gates of p and resize them so that c_p is reduced. This is carried out for all critical output pins in the circuit.

CAM The first step is to identify the candidate gates. These are those gates whose possible shrinking does not potentially increase the circuit delay. For each candidate gate, the size that yields the maximum area reduction without causing the circuit delay to increase is computed. All candidate gates are resized. A delay trace is performed on the entire design. If the circuit delay does not become worse, we stop. Otherwise, we identify an appropriate subset of gates which caused the circuit delay to increase and undo their resizings.

4.3.2 Net Buffering

Fanout optimization refers to buffer insertion and deletion in the circuit either to minimize the circuit delay or to minimize the buffer area, or to avoid pin-overloading. Since we have a globally routed circuit, the structure or topology of each net is known. Then, fanout optimization reduces to placing the right buffers at appropriate points on the nets.

In order to be able to delete a buffer/inverter, we use the concept of an *extended net*, which is a net that passes over buffer cells; the sinks of such a net are either input pins of non-buffer cells or output pads. Figure 3 shows an extended net with root r and four sinks p_1 through p_4. It spans four blocks: (0, 0), (0, 1), (1, 0), and (1, 1). Steiner nodes along with the pins and pads on a net are the candidate locations where new buffers can be inserted.

We have devised a suite of net buffering techniques targeting the four optimization modes. All of them use roughly the following global strategy.

1. Identify candidate extended nets for buffering. These can be either the critical nets (for delay minimization), non-critical nets (for area minimization), or nets whose driver pins violate the load constraint (for pin overloading).

2. For each candidate net, compute the best solution subject to the constraints considering all possibilities for buffer insertion and deletion at each Steiner node. Dummy Steiner nodes may be inserted to meet the goals. Since the buffering choices are exponential, the key to an efficient algorithm is efficient removal of sub-optimal choices at each net node.

3. Select an appropriate subset of nets that show improvement. Buffer them as per their best solutions found in step 2.

4. Do a complete circuit delay trace. If the delay has increased, undo bufferings of the selected nets & stop. Otherwise, go to step 1.

Next we tailor this algorithm to different optimization modes.

UDM: For buffering, UDM is computationally the easiest and fastest of all modes. It is used when the area constraint is not tight. We have implemented Ginneken's algorithm [16] and its enhancements by Lillis *et al.* [9] for a single net. Although the total number of buffering choices is exponential, most of them are provably sub-optimal [16], which leads to a polynomial time exact algorithm for a single-net UDM. There are some practical considerations, however. For instance, each gate has separate rise and fall values for its parameters, such as intrinsic delay α, load coefficient β. Ginneken's algorithm assumes a single value for each parameter. It turns out that separate rise and fall values make the UDM problem for a single net NP-complete [12]. Fortunately, in experiments on several hundred nets, an approximate algorithm that sets the parameter value to the maximum of the rise and fall values was found to be near-optimal [12].

CDM: CDM is much more difficult than UDM, since the area constraint is distributed over several blocks. For instance, the best buffering choice for UDM may not be legal for CDM (i.e., could violate some block-area constraint). Moreover, a sub-optimal buffering choice for UDM may not be sub-optimal for CDM. Because the problem is highly constrained, the number of provably sub-optimal choices goes down, which results in a worst case exponential complexity for a single extended net. However, CDM is more important than UDM, especially when we are close to finalizing the layout. We have proposed a technique to effectively manage the complexity of the problem. It can solve the problem exactly for nets with up to 20 nodes (most of the nets have less than 20 nodes) by keeping track of all the blocks a net passes over and the area penalty incurred by each buffering choice in each block. For larger nets, this technique runs into memory problems and heuristic techniques are resorted to [14].

CAM: CAM is about maximum area recovery without increasing the circuit delay and without violating any load constraints. CAM is of intermediate complexity – between UDM and CDM. Instead of keeping track of the area penalty incurred by a buffering choice in each block the net passes over (as in CDM), it suffices to only store the total area penalty of a buffering choice. Techniques that use smart data structures have been proposed to efficiently detect sub-optimal choices [9, 13].

COF: COF can also be difficult if the area or delay constraints are tight. It uses techniques similar to CDM to generate legal buffering choices for each net. One main difference is that the solution generation proceeds in two phases. In the first phase, only those buffering choices that do not increase the circuit delay are generated. If the pin overloading cannot be fixed with any of these choices, delay-increasing choices are generated in the second phase. The criterion for picking the optimum choice is also different. In the first phase, the buffering choice with the minimum area penalty

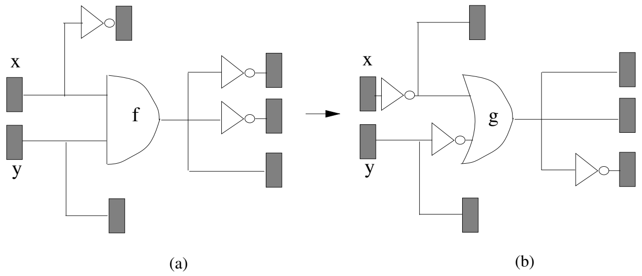

Figure 4: An example of GDM transform: replacing an AND gate with OR & rebuffering incident nets

is picked. In the second phase, the choice with the minimum delay penalty is picked.

4.3.3 Generalized DeMorgan (GDM) Transform

In this transform, we make use of generalized DeMorgan's laws or NN-equivalence to restructure the logic and achieve performance goals. Two functions f and g are **NN-equivalent** if f can be obtained from g by negating zero or more inputs and/or the output of g. For instance, $f = A + B$ is NN-equivalent to $g = A'B$. In this transform, we consider replacing gates in the circuit with their NN-equivalent gates and inserting inverters appropriately at the terminals of the new gates to restore the logic functionality. Thus, this transform can substitute an AND gate with an OR gate (and three inverters) if it is beneficial. In fact, it is more powerful than that. Figure 4 shows how the GDM transform replaces an AND gate f with an OR gate g, and realizes three extra inversions without incurring any penalty by modifying the nets incident to f.

Next we summarize how the GDM transform works. First we identify candidate gates in the circuit on which the transform may be applied, e.g., critical gates for delay minimization. Given such a gate f, the set S of all possible candidate gates g in the library that are NN-equivalent to f are derived. Our goal is to replace f with the gate in S that maximizes the local delay improvement. For each g in S, f is temporarily replaced by g. The set of terminals (input and output pins) T of g that need to be inverted because of NN-equivalence is collected. Although one could simply insert an inverter at each such terminal, our innovation is in constructing, for each terminal p in T, the extended net touching p and invoking the buffering algorithm with additional signal polarity requirements. The goal is to collectively select buffering choices on all these nets incident on T so that the delay improvement at the (critical) gates driving g is maximized. Finally, f is replaced by the gate in S that maximizes this delay improvement.

Since S also includes all resized versions of f (a gate is NN-equivalent to its resized version), GDM transform contains the gate resizing transform. Thus, GDM integrates gate resizing, generalized DeMorgan's laws, and net buffering, and is quite powerful. Although it is less powerful than arbitrary restructuring and remapping, it is more useful in a layout-driven scenario, especially when the layout is almost final. GDM always replaces a gate with one that has the same number of terminal pins. Thus, the global routing patterns and the placement positions in the design are preserved. Only a few buffers need to be inserted or deleted. The current implementation only supports unconstrained delay minimization. We plan to support other modes in near future.

4.3.4 Pin Permutation

Consider a two-input AND gate G with inputs A and B embedded in a circuit. Let the intrinsic delays from the inputs to the output G be $\alpha_{A,G} = 2$ and $\alpha_{B,G} = 1$. Assume the circuit configuration results in the arrival time at A, $\mathcal{A}(A) = 3$, and $\mathcal{A}(B) = 2$ (Figure 5 i). Then, $\mathcal{A}(G) = \max\{3+2, 2+1\} = 5$. However, since G is

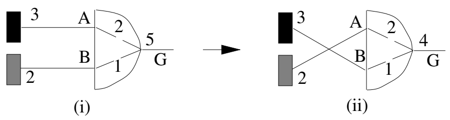

Figure 5: Pin permutation

ex	#cells	#nets	cell area (in BC)	#XxY cuts	o.d. (in ns)
Ckt1	356	409	1567	2x1	7.84
Ckt2	17.1K	26.0K	122.6K	16x32	7.44
Ckt3	40.0K	48.1K	200.2K	16x32	11.38
Ckt4	86.7K	108.1K	381.6K	8x8	18.41
Ckt5	172.2K	210.9K	718.6K	8x8	56.35

1K = 1000, 1 BC = area of the smallest inverter in the library. o.d. = original design delay incorporating layout & wiring details, but before optimization. Elmore delay model used for wiring delays. All benchmarks are in 0.35-μ technology.

Table 1: Benchmark statistics

symmetric with respect to A and B, we can switch the incoming signals without changing the logic functionality, as shown in Figure 5 (ii). Now $\mathcal{A}(G) = \max\{3+1, 2+2\} = 4$. By exchanging signals coming to symmetric input pins of a gate such that the late-arriving signal is applied to an input pin with lower delay, we were able to reduce the arrival time at G by one unit.

Pin permutation makes sense because the technology mapping tool may not have incorporated the best possible pin matching. And even if it had, the arrival times change during layout and pin permutation can come handy. Moreover, it is simple and layout-friendly: it does not disturb any gate placement and affects routing minimally – only the connections to pins of the same gate need to be permuted.

The goal then is: *Given a circuit η, apply pin permutation to the gates of η such that η's delay is minimized.* We omit the algorithm for a single gate; see [3, 2]. For the entire circuit, we apply the single-gate technique to all the critical gates with symmetric inputs. From our experiments, we found that although pin permutation is not as powerful as gate resizing or net buffering, it yields good timing improvements quickly at no cost in area.

5 Experimental Results

In this section, we present experimental results demonstrating the effectiveness of the proposed methodology. We use five industrial circuits, denoted Ckt1 through Ckt5, for our experiments. Characteristics of the benchmarks are given in Table 1. All circuits are optimized by logic synthesis tools, block-placed, and globally routed before being fed to our layout-driven optimization tool. In this paper, since our main focus is on the optimization methodology, we only report the results of invoking layout-driven logic optimizations at a single snapshot in the design flow of Figure 2. The corresponding numbers of horizontal and vertical cuts in the design at this snapshot are shown in Table 1. The interaction between the optimization tool and the min-cut placement and routing tool over successive cut-generation phases is not studied here.

Based on extensive experiments, we have determined that two iterations of net buffering and gate sizing, followed by one or more applications of pin permutation constitutes a fast and effective sequence of optimizations for minimizing circuit delays. Therefore, we use this sequence in all our experiments.[1]

[1]Since our current implementation of GDM does not support CDM or CAM, we did not use it in this set of experiments.

ex	UDM		CDM w/o area rec		CDM w area rec	
	ΔA	ΔD	ΔA	ΔD	ΔA	ΔD
Ckt1	9.64	41.62	21.19	42.81	17.74	43.34
Ckt2	0.22	1.53	2.00	1.53	-0.79	14.92
Ckt3	0.58	23.12	0.63	23.12	-83.37	26.34
Ckt4	0.02	41.71	-0.06	43.15	-12.06	42.33
Ckt5	0.31	27.31	0.28	25.75	-10.01	26.13

ΔA = area increase, ΔD = delay reduction

Table 2: Comparison of UDM, CDM without area recovery and CDM with area recovery. All numbers are percentages.

Our first set of experiments involves unconstrained delay minimization (UDM). In this mode, no area constraints are taken into account when generating and evaluating candidate transforms. However, when actually applying a transformation, if the area in the corresponding block(s) overflows, the transformation is summarily rejected. Columns 2 and 3 in Table 2 give the percentage increase in area and reduction in delay obtained in each circuit with the above sequence of optimizations. The performance of all benchmarks, which had already been optimized by logic synthesis tools, could be further improved at the cost of additional area. The script took 90 minutes for Ckt5 on a 200MHz Ultrasparc with 1.1GB RAM.

To get an idea of the extent to which the circuits are modified, let us look at Ckt2. In UDM mode, 17 buffers are added and 1 buffer is deleted; thereafter 203 gates are resized. Pin permutation does not improve the delay any further, and hence is not applied. Although some area is recovered during resizing of some gates (larger gates replaced by smaller ones), this effect is offset by the area penalty for inserting buffers and resizing other gates.

Let us now apply the same sequence of optimizations as above, but in constrained delay minimization (CDM) mode. Unlike in UDM, area constraints are now taken into account from the beginning, i.e., when generating and evaluating candidate transforms. However, we do not explicitly attempt to recover area in this mode. Columns 4 and 5 in Table 2 show the corresponding percentage increase in area and reduction in delay. The delay has improved in some cases, and has remained the same as in UDM in other cases. The improvements are attributable to the fact that since area constraints are taken into account when generating the candidate transformations, more useful transformations are eventually accepted and applied. For example, in this mode, 320 buffers are added and 222 buffers deleted in Ckt2; also 585 gates are resized. For those circuits in which the delay remains the same as in UDM, the transformations in UDM and CDM appear to be very similar, with minor differences.

Finally, we apply area-recovery by means of gate resizing and net buffering and then apply the above sequence of optimizations in CDM mode. The motivation is (i) to make available more area to enable the optimization algorithms to produce a better solution, and (ii) to let the optimization start from a different point in the optimization space. The results are shown in columns 6 and 7 of Table 2. In general, the performance improves once again, with both factors contributing, we believe. However, the effect of the second factor seems to be more prominent. E.g., in Ckt2, there is hardly any delay improvement without area recovery. With area recovery, it is 14.92%. The delays of all the circuits benefit by area recovery except Ckt4. In Ckt4, the initial area recovery phase frees up 12.84% of the circuit area without increasing the delay. Unfortunately, this also causes the optimization process to start from a point in the optimization space, from where it moves into a local minimum with a worse circuit delay than before. This becomes clear from the inability of the tool to perform as aggressive optimizations as it does without area recovery. This script took about 150 minutes on Ckt5.

In order to better understand the effect of area recovery on performance optimization, we performed another set of experiments. Here, we measured the unused area of each block in the circuit and allowed the optimization tool to use only a small fraction of this area, effectively restricting its choices of "area-hungry" solutions. The results for each of the five benchmarks are presented in Figure 5. The horizontal axis gives the area available in each block for use by the optimization algorithms as a percentage of the total unused area in that block. We will call this the *effective area availability* μ. The vertical axis represents the reduction in critical path delay that can be achieved over the original delay for a given μ.

The following observations can be made from the plots:

1. If μ is very low, it almost always helps to recover area before applying delay optimization. This is intuitive because with insufficient area available for buffers and resized gates, only pin permutation – a relatively less powerful optimization technique – can work.

2. As μ increases to about 5%, the reduction in critical path delay (or the performance of delay optimization algorithms) tends to saturate. Thus, even if more area is made available in each block, the algorithms cannot squeeze too much performance out of the circuits. This implies that in our benchmarks, area availability is not the limiting factor for performance optimization.

3. Recovering the area first changes the circuit slightly and sets the delay optimization process on a different path in the optimization space compared to the case where area is not recovered first. This seems to be an important factor in determining the delay of the final circuit. Since the greedy algorithms used in our tool can get stuck at local minima, the quality of the final solution depends on the starting point in the optimization space. Recovering the area first appears to serve more of that purpose (providing the optimization a better starting point) than providing adequate area for inserting sufficiently large and numerous buffers and/or gates.

6 Conclusions and Future Plan

In this paper we have proposed a constraint-based, layout-driven logic optimization methodology. The approach is to apply a carefully selected set of transforms to the design in a local, incremental, and constraint-based manner between successive cut-generation phases of min-cut based placement- and global routing-refinement, but before the detailed routing stage. Currently it supports four optimization modes: unconstrained & constrained delay minimization, constrained area minimization, and constrained overload fixing. It uses the following four transforms for realizing these modes: gate resizing, net buffering, generalized DeMorgan transform, and pin permutation. The approach has been incorporated in an industrial strength design framework. The results obtained from applying the flow on industrial examples allow us to conclude that: 1) Layout-driven logic optimization as a paradigm is useful. Although the benchmarks were already optimized at the logic level, the layout-driven optimization flow could further reduce both circuit delay and area appreciably. 2) The paradigm of area recovery followed by timing optimization is effective.

In future, we will handle multi-phase clock designs, incorporate a more sophisticated interconnect model, expand the set of constraints & transforms, and address signal integrity issues.

References

[1] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. Multilevel Synthesis Minimizing the Routing Factor. In *DAC*, 1990.

[2] R. Carragher, S. Chakraborty, R. Murgai, M. Prasad, A. Srivastava, and N. Vemuri. Layout-driven Logic Optimization. In *Internal Memorandum*. Fujitsu Labs of America, Oct. 1999.

[3] J. P. Fishburn. LATTIS: An Iterative Speedup Heuristic for Mapped Logic. In *29 ACM/IEEE DAC*, pages 488–491, 1992.

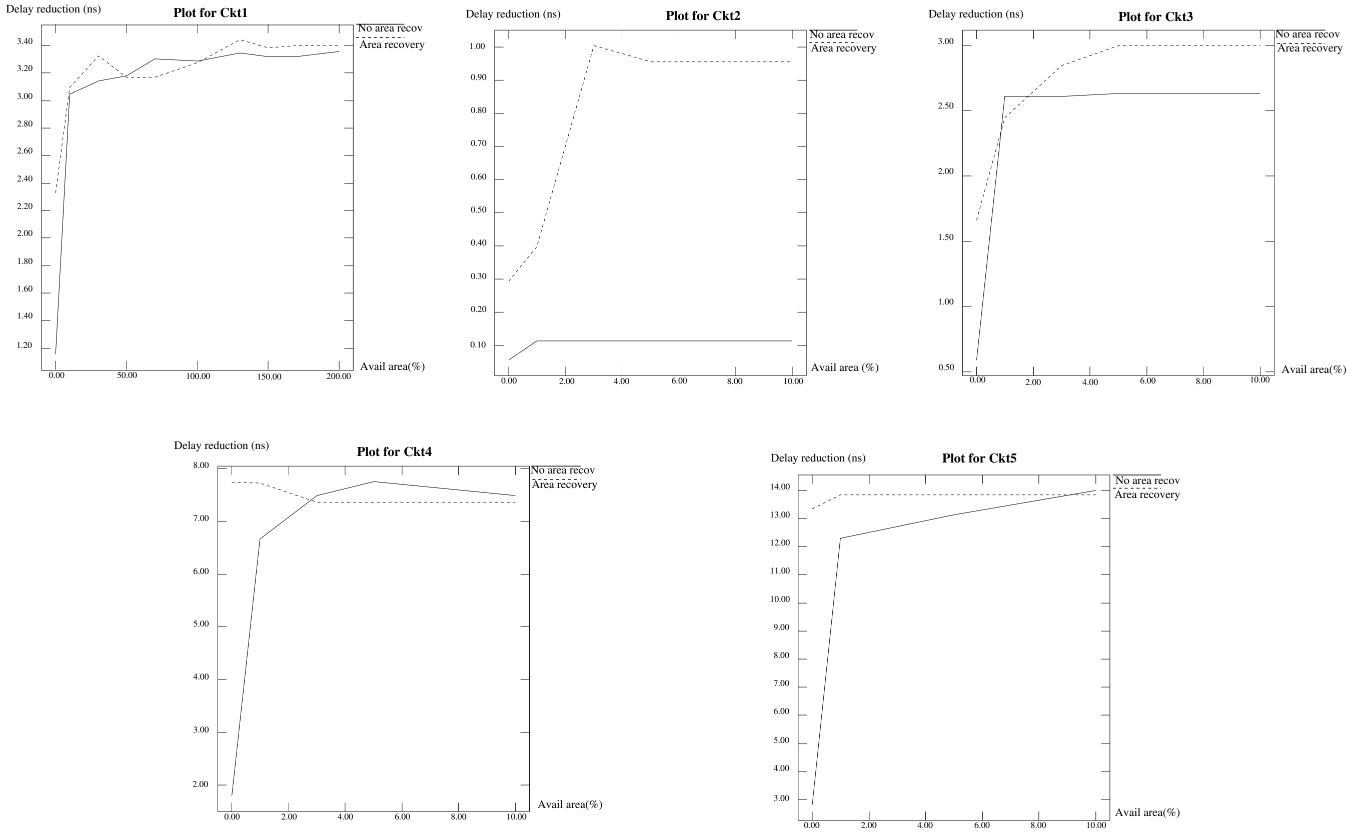

Figure 6: Effect of area recovery on performance optimization

[4] T. Ishioka, M. Murofushi, and M. Murakata. Layout Driven Delay Optimization With Logic Re-synthesis. In *IWLS*, 1997.

[5] Y. Jiang and S. Sapatnekar. An Integrated Algorithm for Combined Placement and Libraryless Technology Mapping. In *ICCAD*, pages 102–105, November 1999.

[6] L. Kannan, P. Suaris, and H. G. Fang. A Methodology and Algorithms for Post-Placement Delay Optimization. In *DAC*, pages 327–332, 1994.

[7] Y. Kukimoto, R. K. Brayton, and P. Sawkar. Delay-Optimal Technology Mapping by DAG Covering. In *DAC*, 1998.

[8] W. N. Li, A. Lim, P. Agarwal, and S. Sahni. On the Circuit Implementation Problem. In *DAC*, pages 478–483, 1992.

[9] J. Lillis, C. K. Cheng, and T. T. Y. Lin. Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model. In *ICCAD*, pages 138–143, 1995.

[10] J. Lou, W. Chen, and M. Pedram. Concurrent Logic Restructuring and Placement for Timing Closure. In *ICCAD*, pages 31–35, November 1999.

[11] J. Lou, A. Salek, and M. Pedram. An Exact Solution to Simultaneous Technology Mapping and Linear Placement Problem. In *ICCAD*, pages 671–675, November 1997.

[12] R. Murgai. Performance Optimization Under Rise and Fall Parameters. In *ICCAD*, 1999.

[13] R. Murgai. Delay Constrained Area Recovery via Layout-driven Buffer Optimization. In *International Conference on VLSI Design*, January 2000.

[14] Rajeev Murgai. Layout-driven Area-constrained Timing Optimization by Net Buffering. In *IWLS*, May 2000.

[15] T. Okamoto and J. Cong. Interconnect Layout Optimization by Simultaneous Steiner Tree Construction and Buffer Insertion. In *ACM/SIGDA Physical Design Workshop*, pages 1–6, 1996.

[16] Lukas P. P. P. van Ginneken. Buffer Placement in Distributed RC-tree Networks for Minimum Elmore Delay. In *ISCAS*, 1990.

[17] M. Pedram and N. Bhat. Layout Driven Logic Restructuring/Decomposition. In *ICCAD*, pages 134–137, 1991.

[18] M. Pedram and N. Bhat. Layout Driven Technology Mapping. In *DAC*, pages 99–105, 1991.

[19] S. Hojat and P. Villarubia. An Integrated Placement and Synthesis Approach for Timing Closure of PowerPC Microprocessor. In *ICCD*, pages 206–210, 1997.

[20] A. Salek, J. Lou, and M. Pedram. A DSM Design Flow: Putting Floorplanning, Technology Mapping and Gate Placement Together. In *DAC*, pages 287–290, June 1998.

[21] A. Salek, J. Lou, and M. Pedram. A Simultaneous Routing Tree Construction and Fanout Optimization Algorithm. In *ICCAD*, pages 625–630, November 1998.

[22] N. Shenoy, M. Iyer, R. Damiano, K. Harer, H-K. Ma, and P. Thilking. A Robust Solution to the Timing Convergence Problem in High Performance Designs. In *ICCD*, Oct. 1999.

[23] G. Stenz, B. M. Reiss, B. Rohfleisch, and F. M. Johannes. Timing Driven Placement in Interaction with Netlist Transformations. In *ISPD*, pages 36–41, 1997.

[24] H. Touati. *Performance-oriented Technology Mapping.* PhD thesis, UC Berkeley, November 1990. UCB/ERL M90/109.

[25] H. Vaishnav and M. Pedram. Routability-Driven Fanout Optimization. In *DAC*, pages 230–235, 1993.