Speeding Up Technology-Independent Timing Optimization by Network Partitioning

Rajat Aggarwal*
Lattice Semiconductor Corporation,
Milpitas, CA
rajat@lscsjc.latticesemi.com

Rajeev Murgai, Masahiro Fujita
Fujitsu Laboratories of America, Inc.
Santa Clara, CA
{murgai,fujita}@fla.fujitsu.com

Abstract

Technology-independent timing optimization is an important problem in logic synthesis. Although many promising techniques have been proposed in the past, unfortunately they are quite slow and thus impractical for large networks. In this paper, we propose DEPART, a delay-based partitioner-cum-optimizer, which purports to solve this problem. Given a combinational logic network that is to be optimized for timing, DEPART divides it into sub-networks using timing information and a constraint on the maximum number of gates allowed in a single sub-network. These sub-networks are then dispatched, one by one, to a standard timing optimizer. The optimized sub-networks are re-glued, generating an optimized network. The challenge is how to partition the original network into sub-networks so that the final solution quality after partitioning and optimization is comparable to that from the timing optimizer. We propose a partitioning technique that is timing-driven and is simple yet effective. We compare DEPART with `speed_up` [21], a state-of-the-art timing optimization tool, and with various partitioning techniques such as min-cut based and region growing, on a suite of large industrial and ISCAS circuits. On more than half of the benchmarks, DEPART yields run-time improvements of 20 to 450 times over a normal invocation of `speed_up` (the overall average improvement being 8 times), without compromising the solution quality much. Min-cut and region growing partitioning schemes, not being timing-driven, perform poorly in terms of the final circuit delay.

1 Introduction

Timing optimization of a digital circuit is an important problem in logic synthesis. It can be performed at different stages of the design process. For instance, *technology-independent optimization* is done before the circuit is mapped on to a technology, whereas *technology-dependent optimization* is carried out during or after mapping. In this paper, we will deal only with technology-independent timing optimization, which addresses the following problem: *Given a Boolean network, the arrival times at the primary inputs, the required times at the primary outputs, and a delay model, obtain a logically equivalent network that meets the required time constraints under the given delay model.*

Many timing optimization techniques have been proposed in literature. They are mostly based on transformations that designers use to speed up circuits. Some of these techniques are *rule-based* such as LSS [8] and SOCRATES [4]. They use a pre-defined set of local transformations based on design style and the target technology to improve the delay. Others use *algorithms* to optimize the circuit. For instance, [9, 22] determine a set of nodes whose restructuring improves the overall circuit performance. The choice of these nodes is made so as to obtain the maximum delay improvement with a minimum area increase. Some algorithmic techniques reduce delay by using network don't cares and permissible functions to re-express node functions using early arriving signals [6]; some others use delay-driven clustering and subsequent collapsing and optimization [23], while others either eliminate long paths that cannot be sensitized by any input vector [5] or reduce the delay of the longest sensitizable path [19].

`speed_up` [21] is a state-of-the-art, topology-based performance optimizer that uses a variety of transformations to optimize the network. These include tree-height reduction [22, 9], timing-driven cofactoring [12], generalized bypass transform [15], timing-driven simplification, timing-driven decomposition, etc. These transformations are local in that they attempt to resynthesize a small part of the network. The best possible timing improvement of each node using these transformations is computed. Finally, a set of nodes on which the transformations should be applied is selected. The selection is based on obtaining maximum timing improvement with minimum area increase.

Although `speed_up` delivers very good-quality solutions (in terms of circuit delay), it turns out that its run-times are exorbitantly large, especially on large circuits and sometimes even on moderately large circuits. For instance, as shown in the column "*Run-time*: `speed_up`" of Table 3, `speed_up` takes 35387 cpu seconds – about 10 hours – to optimize a 1000-gate circuit `ind-1` and over 35 hours for the 1500-gate circuit *table3*. Although `speed_up` generates high-quality solutions, such run-times are simply not acceptable – timing optimization would slow down the entire design cycle.

We propose DEPART, a delay-based logic partitioner-cum-timing optimizer. DEPART assumes that a timing optimizer already exists and then aims to improve the run-time of this optimizer on large or moderately large networks, without modifying the optimizer and without sacrificing the solution quality. It achieves this goal by suitably partitioning the network into sub-networks and dispatching them to the optimizer one by one. The optimized sub-networks are re-glued, generating an optimized network. The challenge is how to partition the original network into sub-networks so that the final solution quality after partitioning and optimization is comparable to that from the timing optimizer. We propose a partitioning technique that is timing-driven and is simple yet effective. The paper is organized as follows. Related work on network partitioning is in Section 2. Section 3 presents our algorithm DEPART. The experimental results are presented in Section 4.

*This work was done when the first author was an intern at Fujitsu Labs of America in the summer of 1996.

0-89791-993-9/97 $10.00 © 1997 IEEE

2 Related Work

Most of the research work in partitioning has targeted the objective of minimizing the number of nets crossing partition boundaries while balancing the sizes of the partitions [13, 2]. This objective is relevant during floorplanning and placement, where tightly-connected components are to be placed in close proximity so as to minimize the total routing length and congestion.

More relevant to us is the problem of partitioning for synthesis. One such work is BEAT_NP [7], a partitioning tool that enables the logic synthesis tool BOLD [3] to handle larger circuits during area optimization. CPU times were reduced by 1 to 3 orders, but were often accompanied by an optimization loss of about 30 to 50%. [10] proposes a network partitioning and resynthesis scheme based on reconvergent fanout regions of the network (*petals* and *corollas*). The goal once again is to resynthesize large networks for smaller layout areas. [17] also targets the same goal. It partitions the network into disjoint sub-networks and optimizes them for minimum area. Nodes in a sub-network are chosen on the basis of common transitive fanins.

Not much work has been done on partitioning aimed at timing optimization of large networks. However, we must mention the problem of *clustering for minimum delay* addressed in [14, 16, 18]: given that an edge crossing cluster boundaries incurs a fixed delay D and that each partition can accommodate no more than M gates, partition the network such that the delay through the network is minimized. This problem is different from the one we are targeting in two respects:

1. the delay D is not relevant to our problem.
2. we are interested in partitioning so that after timing-driven restructuring and optimization, the network delay is minimized. The *clustering for minimum delay* problem does not change the network structure – except possibly replicating some gates.

3 DEPART: DElay optimization using PARTitioning

The main idea of our algorithm DEPART – DElay optimization using PARTitioning – is shown in the flowchart of Figure 1. DEPART reads in an unoptimized network represented in terms of 2-input gates and returns a network optimized for timing, also in terms of 2-input gates. Optimization is achieved by first dividing the given network into sub-networks or partitions, each of which has no more than M (two-input) gates. However, some of our partitioning schemes may initially yield sub-networks with more gates, in which case some gates are temporarily deleted from the partitioned networks to make the networks amenable for further partitioning and size reduction. This further partitioning is achieved by recursively invoking the main algorithm DEPART on the smaller network. The deleted gates are added back to the optimized network returned by DEPART, thus restoring the original logic functionality. On the other hand, if a partitioned network has no more than M gates, the network is handed over to the timing optimizer directly, which attempts to improve the network timing and returns an optimized network. Finally, all the optimized partitioned networks are merged appropriately, generating the optimized network.

DEPART is oblivious to the algorithms employed within the timing optimizer. So, a designer can embed his or her favorite optimizer within DEPART and potentially speed it up. In the current implementation we chose `speed_up`, since it is a state-of-the-art program and generates high-quality solutions.

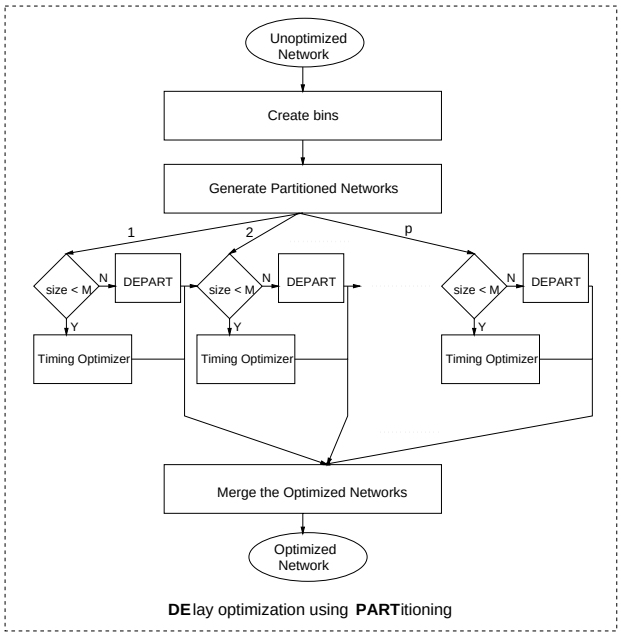

Figure 1: Flowchart of DEPART: DElay optimization using PARTitioning

To see why partitioning can potentially improve the run-time, assume that the worst case complexity of timing optimization is $f(N)$, where N is the number of gates in the unpartitioned network η. Let us also assume that the numbers of gates in the p partitioned networks generated by DEPART are $N_1, N_2, \ldots, N_p$. Ignoring the time taken for computing the partitions (which, as we will see, is negligible anyway), the worst-case complexity for partitioning-based delay optimization, such as DEPART, is $\sum_{i=1}^{p} f(N_i)$. The partitioning-based optimization will be faster if

$$f(N) = f(\sum_{i=1}^{p} N_i) > \sum_{i=1}^{p} f(N_i).$$

In the above, we assumed $N = \sum_{i=1}^{p} N_i$, i.e., the partitions are disjoint. The inequality holds if f is worse than linear. Since timing optimization is NP-hard, f is actually exponential. Then, partitioning-based optimization will definitely be much faster. Typically, practical timing optimization algorithms are not exact but are heuristic in nature. However, they perform tasks that have higher than linear complexities. For instance, `speed_up` [21] computes selection sets through the network and picks the best set by solving a covering problem, which is known to have exponential complexity in the worst-case. So, even for a heuristic optimizer, partitioning should improve the run-time.

With partitioning-based optimization the danger, however, is that we may lose out on optimization, because the optimizer sees only a part of the entire network at any time. The crucial issue, then, is *how to partition the network such that the final solution quality (primarily timing, but also area) after partitioning and optimization is comparable with that of a standard optimizer*, say `speed_up`. Partitioning algorithms in DEPART are timing-driven, i.e., they cluster network nodes with a view to rendering the forthcoming performance optimization effective – from the point of view of both circuit performance and CPU time. DEPART first creates bins; each bin contains a subset of primary outputs and their transitive fanin nodes. From the bins, partitioned networks (or sub-

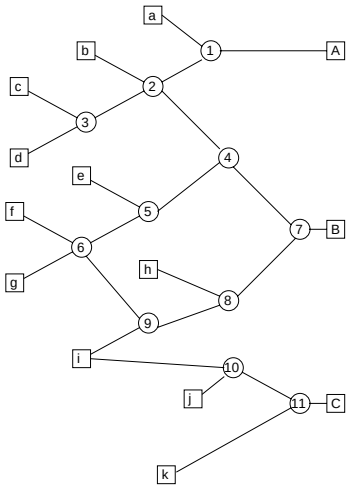

Figure 2: Example network consisting of three primary outputs A, B, and C, and primary inputs a through k.

networks) are generated and sent to the timing optimizer. The resulting optimized sub-networks are glued together yielding the optimized network. These steps are described in the next few subsections.

We will explain this algorithm with the help of a running example – the network η of Figure 2. η has eleven primary inputs (PIs) a through k, three primary outputs (POs) – A, B, C, and eleven internal nodes (or gates) 1 through 11. Although not shown, the signal flow is from left to right.

3.1 Creating Bins

We divide the network into bins, each containing some primary outputs and their transitive fanins (TFIs). The crucial point is to decide which outputs should be grouped together in a bin. We developed three strategies.

1. **Minimum bins**: In this strategy, the outputs are grouped based on the number of TFI nodes they share. Outputs that share many TFI nodes are placed in one bin. Recall that no partition (and hence bin) should have more than M gates. Our goal is to minimize the number of bins needed to accommodate all the primary outputs and their transitive fanins, without violating the constraint M. Since each bin would correspond to a partitioned network, minimizing bins minimizes the number of partitioned networks resulting in better optimization.

   The problem of grouping outputs in minimum bins subject to the size constraint M is similar to the **bin-packing problem** [11], in which items with weights are to be packed in minimum number of uniform bins each of capacity C such that the sum of the weights of the items in a bin is at most C. Apart from some minor technicalities, the correspondence between the two problems is quite straightforward. Each primary output o_i along with its transitive fanin is considered an item, whose weight $w(o_i)$ is the number of gates in the transitive fanin of the output. The capacity C of each bin is M, which means that no more than M gates can be accommodated in each bin. The goal is to use fewest bins to store all the items.

   On the outputs whose TFIs have at most M gates, we apply the standard best-fit decreasing heuristic [11]. Each of the remaining outputs is stored in a separate bin.

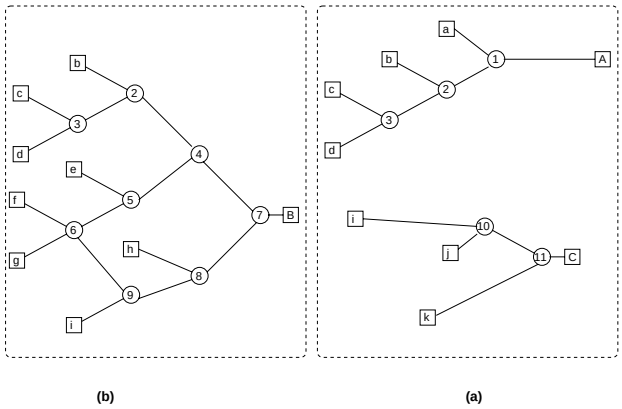

Figure 3: Bins created with $M = 5$ and `minimum bins`

We illustrate bin creation on the example network of Figure 2. The transitive fanins of A, B, and C have 7, 16, and 5 nodes respectively, out of which 3, 8, and 2 are internal nodes (or gates). Let M be 5. First, we create special bins for outputs with weights more than 5. The only such output is B, and we obtain the bin (b) in Figure 3. We sort the remaining outputs, A and C, on the number of TFI gates. We obtain the sorted list $L = \{A, C\}$. A and its TFI are placed in a newly created bin, (a). It turns out that C can also fit in the same bin. Note that we are considering gates and not PI/POs while checking the fitting of items into bins. The bin (a) is now full to capacity.

2. **Criticality information**: The basic idea is to group **critical** primary outputs in fewest bins. A critical output is one that needs to be speeded up in order to improve the performance of the circuit (we will make the notion of criticality more precise shortly). Non-critical primary outputs are not sent for optimization, thus reducing the total run-time tremendously.

   First, a delay trace is carried out on the network to determine the arrival times, required times, and slacks at all the nodes of the network. The arrival times of the primary inputs can be set by the user, 0 being the default. If no required times are set at the outputs, they are set to be the maximum arrival time of an output. A primary output is critical if its slack is at most t% of the maximum arrival time of some network output, where t is a user-specified number. In case no required times are set at the outputs, we can alternatively define an output to be critical if its arrival time is at least $(100 - t)$% of the maximum arrival time. If t is 0, only the latest arriving outputs are critical. If it is 100, all the outputs are critical.

   After the delay trace, non-critical outputs are determined and ignored from consideration. The critical outputs are sorted into list L, the most critical output being first. Then, a bin-packing-like method is used to group the sorted critical outputs and their TFIs into bins. The first output from L is placed in a new bin. Thereafter, at any step, the next output is fetched from L and considered for placement in the same bin as the last output. Thus, an attempt is made to place critical outputs in fewest bins solely based on criticality information (and not on node sharing, which was the case in the `minimum bins` approach).

   Let us consider the example of Figure 2. Assume that each gate has a delay of 1 unit, and that all inputs arrive at time

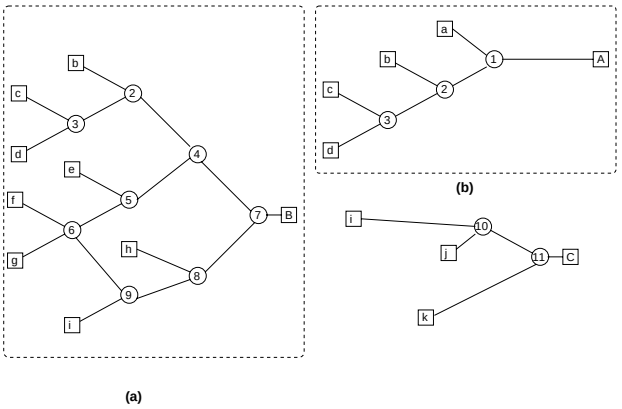

Figure 4: Bins created with $M = 5$ and `criticality`

0. After delay trace, we find that the arrival times of A, B, and C are 3, 4, and 2 respectively. If $t = 25$, any output with arrival time at least 75% of 4 (= 3) is critical. So A and B are critical and C is not. Two bins are created, one with B and the other with A. This is shown in Figure 4. C is ignored from consideration.

3. **Minimum Critical Bins**: This strategy is a cross between the last two. As in `criticality`, non-critical outputs are removed from consideration. Then critical outputs are placed in bins using the strategy employed in `minimum bins`. Note the difference between this strategy and `criticality`. In `criticality`, critical outputs are placed solely on the basis of criticality information, whereas in `minimum critical bins`, they are placed solely on the basis of TFI information.

   For our example, `minimum critical bins` creates the same bins as created by `criticality` in Figure 4. However, the order in which the bins will be processed will be different. We will explain this later.

Each bin generates a partitioned network that is sent to the timing optimizer. The next subsection describes how partitioned networks are generated from bins.

3.2 Generating Partitioned Networks From Bins

Bin-creation phase generates information about the primary outputs and nodes present in each bin. This information is used to derive partitioned networks. Bins, in general, share nodes. TFI of an output that belongs to bin A may share some nodes with TFI of an output belonging to bin B. While creating networks from bins, one has to decide whether these common nodes should be replicated in each network or just be present in one of them. We propose three ways to handle this:

1. **No-Overlap**: In this option, no overlap is allowed among partitioned networks, except for primary inputs. Assume that bins are traversed in some order (we will address the ordering issue soon). If bin X has an internal node (or gate) n that is present in bin Y earlier in the order, but not in any bin prior to Y, n will be included in the network for Y, but not in the network for X. Instead, a *dummy primary output* fed by n is introduced in the network corresponding to Y. Also, in the network corresponding to X, a *dummy primary input* is added. However, not every common node n is converted into dummy input and output; only the boundary nodes, as explained below.

   Figures 5(a) and 5(b) show the networks generated from the bins of Figure 3 using `no-overlap`. For the sake of the example, assume that the bins are processed in the order 3 (a), 3 (b). Creating the network from 3 (a) is straightforward – include all the nodes of bin 3 (a) in the partitioned network. Next, 3 (b) is processed. Primary inputs b, c, d, and nodes 2 & 3 are common to bins 3 (a) and 3 (b). However, none of the nodes b, c, d, and 3 fan out to a node that is in bin 3 (b) but not in bin 3 (a). So b, c, d, and 3 are not made primary inputs for the network 5 (b). However, node 2 fans out to node 4, which is in bin 3 (b) and not in 3 (a). So a dummy primary output $2A$ is introduced in network 5 (a). Similarly, dummy primary input $2a$ is added to network 5 (b). It is implicit that $2A$ and $2a$ are essentially the same signal. Adding $2A$ to the network 5 (a) ensures that during timing optimization of the network 5 (a), node 2 is retained, so that it can be later composed into the optimized network 5 (b) at the primary input $2a$.

   Although `no-overlap` does not carry any area penalty, it can break a critical path several times, potentially resulting in a loss of optimization. This fact motivated us to devise the bin creation heuristic `criticality` in its current form. By sorting the outputs by criticality and grouping the highly critical ones in the first few bins, the algorithm attempts to retain complete input to output (I/O) paths of more critical outputs in a single sub-network.

2. **Complete-Overlap**: This option lies on the other end of the spectrum. Nodes common to different bins are replicated in the corresponding networks. Each partitioned network thus retains all the nodes of the corresponding bin. Referring to the running example, the networks corresponding to the bins of Figure 3 are identical to those of Figure 3.

   This option allows maximum flexibility in terms of timing optimization (modulo the loading effects), since complete I/O paths are available to the optimizer. However, it typically results in a huge area penalty.

3. **Critical-Overlap**: Here only critical nodes are replicated. The intuition is that the timing optimizer can optimize better when complete critical paths are available. Also, by restricting replication to critical nodes, area penalty is reduced. As for the non-critical nodes, they are not duplicated, but are converted to *dummy primary inputs* and *dummy primary outputs* appropriately as described in `no-overlap`.

Processing Order for Bins: In order to generate networks from bins, bins have to be processed in some order. This order can have a great impact on the kinds of networks generated and hence the run-time and solution quality.

In `minimum bins` and `minimum critical bins` options, bins are traversed in the order of increasing sizes (i.e., used capacities). To understand why, consider our example. In `minimum bins`, we would first process bin (a) of Figure 3, since it has only 5 gates, as compared to the special, large bin (b), which has 8 gates. Next, we come to bin (b). With the `no-overlap` option, the gates common to the two bins are excluded from the network generated from bin (b). As shown in Figure 5 (b), this network has 6 gates. This ordering scheme then helps to reduce the sizes of the networks that would have been otherwise large.

In `criticality`, bins are traversed in the order they were created – the most critical first. For our example, `minimum critical bins` and `criticality` generate identical bins. However, in `minimum critical bins` the bin with output B, being larger, is processed after the bin with output A. This implies that with the

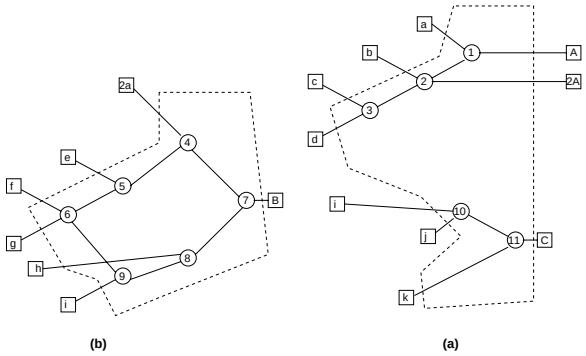

Figure 5: Partitioned networks generated with `no-overlap`

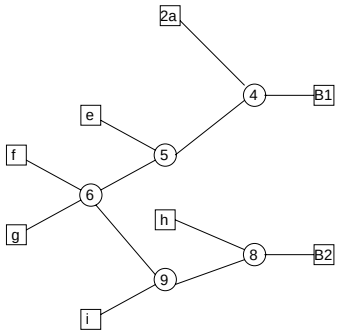

Figure 6: The network generated by the second level partitioner after deleting the primary output B.

`no-overlap` option, nodes 2 and 3 will be deleted from the network for B. With `criticality`, B will be processed first (since it is more critical), and nodes 2 and 3 will be deleted from the network of A.

3.3 Optimizing the Partitioned Networks

After partitioned networks have been created, they are sent for timing optimization. However, it may happen that some partitioned networks have more than M gates. For instance, the network of Figure 5 (b) has 6 nodes, and further partitioning is needed to reduce the size to no more than $M = 5$. We incorporate this by defining a recursive optimization-partitioner. When the number of internal nodes, S, of the partitioned network is no more than M, we enter the **base level** optimization: partitioned network is sent directly to the standard timing optimizer such as `speed_up`. However, when S is greater than M, the partitioned network is sent to a **second level partitioner-cum-optimizer**. This partitioner-cum-optimizer program appropriately selects a primary output P from the partitioned network,[1] deletes P, converts all its immediate fanins to primary outputs, and makes a recursive call to the main partitioning algorithm DEPART on the network thus obtained.

The intuition behind this strategy is the following. First recall that the network at this stage is in terms of 2-input gates and therefore the number of fanins of P is 2. After deleting P, these fanins are converted into primary outputs. If the sizes of the TFIs of each of these outputs are roughly the same ($\sim S/2$), and furthermore if $S/2 \leq M$, the outputs will satisfy the capacity constraint M during the first level of recursion itself. If $S/2 > M$, further recursion may be needed, but the process will quickly lead to partitions of sizes at most M.[2] An advantage of this method is that a path in the original network completely resides in the new network, except perhaps for the terminal node, if that happens to be P.

The partitioned network in Figure 5(a) has 5 nodes. Since $M = 5$, this sub-network is directly sent to the timing optimizer (the base case). However, the network in Figure 5(b) has 6 nodes, and is sent to the second level partitioner-cum-optimizer. The second level optimizer selects the primary output B, deletes it, and converts its immediate fanins 4 and 8 to primary outputs $B1$ and $B2$. Figure 6 shows the network thus generated. This network has 5 gates, and when sent recursively to DEPART, it is immediately sent to the timing optimizer. However, if M was 4, in the recursive call to DEPART, the network of Figure 6 would go through the steps of bin creation and network creation. Note that each of the two outputs $B1$ and $B2$ has 3 internal nodes. So two bins will be generated. Both corresponding networks will satisfy the M constraint and hence would be sent to the timing optimizer.

Now we explain how timing information is propagated between the partitioned networks during optimization. Assume that the original network η is partitioned into networks $\eta_1, \eta_2, \ldots, \eta_p$. Also assume that the partitioned networks are sent for optimization in the same order, i.e., η_1 first and η_p last. Say at some stage, η_i is sent for optimization. Then, after optimization, the arrival times of the primary outputs of η_i may change. If some primary output of η_i is a dummy output, it feeds dummy input(s) of other partitioned network(s), say η_j. The new arrival time is used to update arrival time of the corresponding dummy primary input(s), so that when η_j is optimized later (assuming $j > i$), the timing information used in optimization is the updated one. In fact, in DEPART, we make sure that if $i < j$, no dummy primary input of η_i is connected to a dummy primary output of η_j. Thus, the arrival time information of η_i depends only on the arrival times of the real primary inputs of η and possibly of the dummy primary outputs of networks $\eta_k, k < i$.

3.4 Combining the Optimized Partitioned Networks

After all the partitioned networks have been optimized, they are merged to form an optimized network having the same functionality as the original network. The optimized partitioned networks are merged by patching up the corresponding dummy primary outputs and dummy primary inputs. For instance, if (a) and (b) of Figure 5 are the optimized partitioned networks, after merging we obtain the optimized network shown in Figure 2. The only dummy primary input-output pair is $(2a, 2A)$. At the time of merging, the fanin of $2A$, node 2, is made to fan out to all the fanouts of $2a$, which, in this case, is the single node 4. Both $2a$ and $2A$ disappear after merging.

Thus we obtain a functionally-identical, timing-optimized circuit. Next, we report the experimental results on a set of industrial and ISCAS benchmarks.

4 Experimental Results

DEPART is implemented in the sis framework [20], the logic synthesis system from the University of California, Berkeley. In the

[1] P is the latest arriving output.

[2] It can happen that the sizes of the two TFIs are disparate for many consecutive recursion calls. In that case, we limit the maximum depth of recursion and then call the timing optimizer on the network at the final level.

Bench.	#I/P	#O/P	#Lev.	#Lit.
Ind-1	8	8	37	1074
Ind-2	8	8	23	435
C880	60	26	35	684
C6288	32	32	92	4962
C7552	207	108	35	3976
C3540	50	22	40	2222
seq	41	35	17	2937
apk2	1263	1218	33	22096
alu4	14	8	28	1745
apex5	114	85	24	1346
C5315	178	123	32	2911
C2670	233	139	20	1379
table3	14	14	55	1456
apex3	54	50	14	2750
des	256	245	22	5889
pair	173	137	28	2949

Table 1: Characteristics of the benchmarks used as test inputs

current implementation, DEPART invokes `speed_up` on each subnetwork. Since `speed_up` lets one specify the delay model to be used during optimization, so does DEPART. Currently, DEPART uses the *unit delay model*, which assigns a delay of one to each 2-input gate and thus estimates the circuit delay to be the number of levels.[3]

The benchmarks used in the experiments come from two sources: some are combinational ISCAS benchmarks and the rest (i.e., Ind-1, Ind-2, apk2) are industrial circuits. We first optimized the benchmarks for minimum area (i.e., factored form literals) using the standard sis optimization script, `script.rugged` [20], twice with a timeout option of 1 hour each time and then applied `eliminate -1; speed_up -i` to obtain networks that would be starting points for all the experiments. `eliminate -1` collapses all nodes that do not save any literals into their fanouts. `speed_up -i` performs a timing-driven decomposition on each network node and produces a network that has either inverters or two-input gates. Table 1 shows the numbers of primary inputs, primary outputs, levels, and literals in these networks. With the exceptions of Ind-2 and *C880*, the criterion for selecting a benchmark was that it should have at least 1000 two-input gates (roughly equal to the number of literals).

DEPART uses several control parameters that influence the final solution quality and run-time, such as the maximum size of a partitioned network M, the bin creation heuristic, degree of overlap between partitions, etc. We did extensive experimentation to determine the best values for them. The best values were found to be $M = 200$, bin creation – a combination of `criticality` and `minimum bins`, overlap among partitioned networks – `no overlap` [1].

4.1 Comparison with Other Schemes

In this section we compare and analyze results of DEPART with `speed_up` (invoked on the entire network), and two partitioning-based schemes:

1. hMETIS, a state-of-the-art min-cut-based partitioning algorithm [13], and
2. a region growing algorithm based on connectivity information.

None of these partitioning schemes are driven by timing considerations. A comparison with them would highlight the significance of a timing-driven partitioning scheme such as DEPART.

hMETIS is a multi-level min-cut partitioning algorithm. It is a very fast and accurate scheme that partitions a netlist into a prespecified number of approximately equal-sized clusters and minimizes the total number of nets crossing the cluster boundaries. It produces k partitions using a bisection algorithm recursively. hMETIS first coarsens the netlist in multiple levels to get a smaller representative netlist. A partitioning solution is calculated on the smaller netlist which is then refined in the uncoarsening phase. These coarsening and uncoarsening phases allow hMETIS to obtain a global picture, resulting in a high-quality solution. These phases also help in reducing the run-time, since most of the computation is performed on the smaller netlist.

We implemented a region growing algorithm, using a recursive bisection technique. The bisection process starts from a randomly selected node and grows in a breadth-first fashion a region around it of nodes tightly connected to it [13] until half of the nodes of the netlist are in this region. The nodes belonging to the grown region are assigned to the first part and the rest to the second part. The process is repeated on the two parts until k partitions are obtained.

Timing optimizer (`speed_up`) is invoked on each cluster obtained by either partitioning algorithm (hMETIS or region growing algorithm). Timing information (such as arrival times) is updated and propagated to other clusters after each invocation. This is similar to the timing optimization step in DEPART. The difference between DEPART and these partitioning techniques is the target cost function. As described earlier, DEPART attempts to generate partitions with a view to maximizing the optimization potential of the timing optimizer. These techniques, however, do not pay any attention to timing. Instead, they either minimize the total number of nets crossing the cluster boundaries (hMETIS) or maximize the connectivity within each cluster (region growing).

`speed_up` is invoked with unit delay model (so it minimizes the number of levels) and depth of collapse of transitive fanin region equal to 4.

For hMETIS and region growing, we set the number of clusters to the number of nodes divided by M (=200).

Tables 2 and 3 show comparison of various techniques. For each benchmark, numbers of levels and literals (a measure of network area) before and after timing optimization (with and without partitioning) are reported. For each technique, run-times on Sun SPARCstations 20 are reported in seconds in Table 3.

From Table 2, we deduce that with respect to number of levels, on average, DEPART is about 6.6% worse than `speed_up`[4] and in terms of literals, it is marginally better. However, Table 3 shows that on average, it is more than 8 times faster than `speed_up`. It is interesting to note that `speed_up` takes about 10 hours on Ind-1, which has about 1000 2-input gates. DEPART, however, takes about 15 minutes (which is 40 times less) and generates same number of levels and fewer gates! On *table3*, DEPART is more than 450 times faster than `speed_up` (`speed_up` takes 35.5 hours and DEPART only 4.7 minutes) – with only 7% penalty in the number of levels. Run-time improvements by factors of 20 to 40 can be seen in many other examples, such as Ind-2, *C880*, *C3540*, *seq*, *alu4*,

[3] Singh concludes in [21] that optimized networks generated using unit delay model have post-map delays comparable with delays of networks generated using more sophisticated models, such as unit fanout model, library model, etc. We reached the same conclusion using a similar set-up and a 0.5-micron technology library [1].

[4] "% Change" in the table is with respect to the original network (the column "Org"). The figure 6.6%, although not in the table, is obtained from a pairwise comparison between `speed_up` and DEPART.

Bench.	Levels					Literals				
	Org	speed_up	m-c	r-g	DEPART	Org	speed_up	m-c	r-g	DEPART
Ind-1	37	30	34	37	30	1074	1383	1117	1074	1249
Ind-2	23	18	22	22	20	435	553	451	447	465
C880	35	18	21	35	21	684	1005	1007	684	913
C6288	92	70	78	92	75	4962	5582	6139	4962	6367
C7552	35	21	29	35	22	3976	4505	4566	3976	4577
C3540	40	31	33	40	33	2222	2414	2331	2222	2382
seq	17	14	15	17	15	2937	2964	2940	2937	3003
apk2	33	16*	32	33	16	22096	22094*	22375	22096	21666
alu4	28	25	28	28	26	1745	1937	1745	1745	1917
apex5	24	14	16	24	14	1346	1580	1954	1346	1580
C5315	32	21	27	32	25	2911	3387	3067	2911	3099
C2670	20	17	18	20	17	1379	1469	1405	1379	1482
table3	55	41	52	54	44	1456	2287	1493	1458	2750
apex3	14	12	13	14	12	2750	2790	2770	2750	2810
des	22	19	20	22	20	5889	6043	6508	5889	6223
pair	28	17	17	28	20	2949	3282	3950	2949	3158
% Change		-27.45	-14.81	-0.40	-22.87		16.36	12.49	0.17	15.78

Org `original benchmark -- before optimization`
m-c Min-cut approach of hMETIS
r-g Region Growing

Table 2: Comparing speed_up, min-cut based hMETIS, region growing and DEPART. *: the best network at the time of abortion, which was done after more than 2 days. % Change is with respect to "Org".

C5315, *C2670*, and *pair*. This improvement is achieved at the cost of a very small delay penalty, maximum penalty being about 15%. Three factors contribute towards such an advance:
1) the inherent run-time advantage of partitioning – this was discussed in Section 3,
2) Partitioning is timing-driven:

a. Partitions are formed with timing optimization in mind.
   - Outputs are grouped in a bin, keeping in mind gate-sharing (to save area) and criticality (for timing).
   - Complete I/O paths are handed to the optimizer (as much as possible).
   - Non-critical outputs are ignored.

b. Updated timing information is propagated from an optimized partition to partitions yet to be optimized.

3) careful parameter-tuning in DEPART.

We also noted that the CPU time taken only by the partitioning algorithm (i.e., not including the time taken by speed_up for timing optimization) is negligible as compared to the time spent in calls to speed_up – it is about 4% of the total time.

Sometimes, though rarely, DEPART takes appreciable run-time. For our benchmark suite, it happens only on one example, *apex3*; DEPART takes about 16 hours. We believe this is due to speed_up (which is invoked from within DEPART many times) taking long time to choose selection-sets for applying the transformations. speed_up builds a binary decision diagram (BDD) to solve the selection-set problem and the size of the BDD cannot be predicted before-hand.

It is interesting to study the relative performance on the largest circuit in the benchmark suite: apk2. First of all, speed_up did not terminate in more than 2 days. So, for comparison, we decided to select the final network generated by speed_up at the time of abortion (speed_up is an iterative algorithm and at the end of each iteration one can save the best network generated so far). The run-time reported is the time taken to generate the final network when the program was aborted. DEPART and speed_up yield the same number of levels. DEPART has about 5% area penalty, but is about 15 times faster than speed_up. We also observed that speed_up was using more than 1000 MB of memory at the time of abortion. This was causing thrashing and a slow-down of the entire optimization process.

Note that DEPART makes run-time calls to speed_up, without using any information about the algorithms employed within it.

The overall performance of min-cut-based hMETIS is far from satisfactory. On average, it results in 19.8% more levels than speed_up and about 12.9% more levels than DEPART. For the benchmark apk2, DEPART reduced the number of levels from 33 to 16, whereas hMETIS reduced it only to 32! Thus there is a strong case for using a timing-driven partitioner. The quality of the resulting circuit can be quite bad with min-cut. Except for one example (C7552), hMETIS run-times are comparable to DEPART's.

The results of region growing algorithm are very poor. Except for two benchmarks, namely, Ind-2 and *table3*, there was no improvement on any of the other benchmarks. Even on Ind-2 and *table3*, the improvements were marginal. The reason behind this is that partitioning, not being timing-driven, was unsuccessful to give complete I/O paths to the timing optimizer.

We also compared DEPART with the *clustering for minimum delay* algorithm, which was mentioned in Section 2. The implementation of [16] is not very practical, since it results in about 30 to 70% gate-replication. We modified this algorithm to generate clusters without replication and redefined the meaning of the inter-cluster delay D to be more suitable for our application. It turns out that DEPART is twice as fast and generates faster and smaller circuits. Due to space limitations, we omit the details; they can be found in [1].

Finally, we mapped unoptimized and optimized circuits on to Fujitsu's 0.5-micron technology library. The technology mapper was run in delay-minimization mode. Although we do not report the mapped results due to space constraint, we found that on average, the delay through a mapped network generated by DEPART is about the same as through that generated by speed_up – about 1% more [1].

Bench.	Run-time			
	speed_up	m-c	r-g	DEPART
Ind-1	35387.0	173.1	3.3	941.4
Ind-2	3026.6	30.0	2.9	114.2
C880	13367.7	327.2	1.8	388.4
C6288	12728.3	1655.2	55.3	2151.5
C7552	2522.5	56572.0	38.1	459.6
C3540	22671.7	363.6	11.1	336.4
seq	18557.0	449.9	19.3	626.7
apk2	76922.2*	6931.3	2029.1	5001.5
alu4	7698.9	108.9	7.8	333.0
apex5	343.2	781.0	5.7	137.2
C5315	6377.5	570.0	20.0	175.7
C2670	3179.8	148.4	6.9	101.3
table3	128323.4	136.8	5.9	282.9
apex3	127026.9	817.9	16.6	57455.9
des	4490.1	4576.0	47.9	1879.2
pair	4087.5	834.8	20.8	186.1
% Change	209.07	-36.53	-98.80	-73.73

m-c Min-cut approach of hMETIS
r-g Region Growing

Table 3: Comparing run-times of speed_up, min-cut based hMETIS, region growing, and DEPART. The %Change is with respect to the average value of run-times of the four techniques. *: the best network at the time of abortion, which was done after more than 2 days.

Future work is in at least two directions:
1) The performance of DEPART is far from satisfactory on *apex3* – it takes more than 15 hours to finish. We need to investigate this further and come up with a better methodology.
2) DEPART is built on top of a timing optimizer. Further gains in run-times can be accrued if the underlying timing optimizer can be speeded up.

References

[1] R. Aggarwal, R. Murgai, and M. Fujita. Speeding Up Technology-Independent Timing Optimization. In *Fujitsu Labs of America Internal Memorandum*, September 1996.

[2] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[3] K. Bartlett, D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, M. Moceyunas, C. Morrison, and D. Ravenscroft. Bold: A multi-level logic optimization system. In *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[4] K. Bartlett, W. Cohen, A. J. De Geus, and G. D. Hachtel. Synthesis of Multi-level Logic Under Timing Constraints. In *IEEE Transactions on Computer-Aided Design*, October 1986.

[5] H. Chen and D. Du. Circuit Enhancement by Eliminating Long Paths. In *Proceedings of the Design Automation Conference*, pages 249–252, 1992.

[6] K. C. Chen and S Muroga. Timing Optimization for Multi-Level Combinational Circuits. In *Proceedings of the Design Automation Conference*, pages 339–344, 1990.

[7] H. Cho, G. Hachtel, M. Nash, and L. Setiono. BEAT_NP: A Tool for Partitioning Boolean Networks. In *Proceedings of the International Conference on Computer-Aided Design*, pages 10–13, 1988.

[8] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L.Trevillyan. LSS: A system for Production Logic Synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.

[9] G. De Micheli. Performance-Oriented Synthesis of Large-Scale Domino CMOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):751–765, 1987.

[10] Sujit Dey, Franc Brglez, and Gershon Kedem. Corolla based Circuit Partitioning and Resynthesis. In *Proceedings of the Design Automation Conference*, pages 607–612, 1990.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., NY, 1979.

[12] P. Gutwin and P. C. McGeer. Delay predictor for technology-independent logic equations. In *Proceedings of the International Conference on Computer Design*, pages 468–471, 1992.

[13] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Hypergraph Partitioning: Application in VLSI Domain. In *Proc. ACM/IEEE Design Automation Conference*, 1997.

[14] E. L. Lawler, K. N. Levitt, and J. Turner. Clustering to Minimize Delay in Digital Networks. In *IEEE Transactions on Computers*, pages 47–57, January 1969.

[15] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni. Performance Enhancement through the Generalized Bypass Transform. In *Proceedings of the International Conference on Computer-Aided Design*, pages 184–187. IEEE, 1991.

[16] R. Murgai, R. K Brayton, and A. Sangiovanni-Vincentelli. On Clustering for Minimum Delay/Area. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[17] Y. Nakamura and T. Yoshimura. A Partitioning-based Logic Optimization Method for Large Scale Circuits with Boolean Matrix. In *Proceedings of the Design Automation Conference*, pages 653–657, 1995.

[18] R. Rajaraman and D. F. Wong. Optimal Clustering for Delay Minimization. In *Proceedings of the Design Automation Conference*, pages 309–314, June 1993.

[19] A. Saldanha, H. Harkness, P. C. McGeer, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance Optimization Using Exact Sensitization. In *Proceedings of the Design Automation Conference*, pages 425–429, 1994.

[20] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis.* Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[21] K. J. Singh. *Performance Optimization of Digital Circuits.* PhD thesis, UC Berkeley, December 1992.

[22] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 282–285. IEEE, 1988.

[23] H. J. Touati, H. Savoj, and R. K. Brayton. Delay Optimization of Combinational Logic circuits by Clustering and Partial Collapsing. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191. IEEE, November 1991.