VLSI DESIGN
1998, Vol. 7, No. 1, pp. 1–13
Reprints available directly from the publisher
Photocopying permitted by license only

Clustering Network Modules with Different Implementations for Delay Minimization

DIMITRIOS KARAYIANNIS[a] and SPYROS TRAGOUDAS[b,*]

[a]*Viewlogic Systems Inc., Fremont CA 94538-6530;* [b]*Computer Science Department, 2130 Faner Hall, Southern Illinois University at Carbondale, Carbondale, IL 62901*

In recent years there has been an extensive interest in clustering the modules of a network so that the maximum delay from any *primary input* to any *primary output* is minimized [8, 7, 6]. Clusters have a maximum capacity and modules may have different implementations. All existing CAD frameworks initially select an implementation of each module, and at a later stage they cluster the modules. We present an approach that clusters the nodes, while considering their alternative implementations, so that we further minimize the maximum delay after the clustering. Our approach is based on optimal algorithms for restricted versions of this complex problem in circuit design, and outperforms the conventional approach, which first obtains an implementation for each circuit module without considering clustering and then, in a later step, performs clustering.

Keywords: Clustering, circuit implementation, delay minimization, NP-completeness, algorithms

1. INTRODUCTION

The problem of partitioning the modules of a combinational circuit into clusters that have a uniform capacity bound B, has been extensively studied [3, 8]. Each circuit module has an integer delay and an integer area associated to it. Furthermore, we assume an inter-cluster delay D. The notion of inter-cluster delay is understood as follows. If two nodes, u and v, are placed into two different clusters, then the delay of the interconnection from u to v is incremented by D units. Recently, much effort has been devoted into clustering the modules so that the maximum delay from any *primary input* to any *primary output* is minimized under the above constraints [6, 7].

This type of partitioning problem occurs either at the system level or at the board level [8]. At system level partitioning every cluster corresponds to a Printed Circuit Board (PCB) whereas at the board level to a chip. In the former case, the inter-cluster delay amounts to the off-board delay along the system buses. The system buses are very slow since certain criteria must be met so that different boards can share the same bus [8]. In board level partitioning, the off-chip delay is also much larger

*Corresponding author.

than the on-chip delay which is practically encountered on the modules. In both cases, the value of B is often a constant.

In such timing-driven partitioning approaches the total area of the circuit after the clustering is not considered to be the primary restriction. Thus, the optimal polynomial time approach in [7] allows for modules to be replicated in order for the maximum delay to be minimized. Although optimal, the algorithm in [7] does not take into consideration the fact that, in reality there may be more than one implementations per module in a given library [1, 2]. (A module at the board level partitioning corresponds to a gate and at the system level partitioning to a chip). Different implementations of the same module may have different areas and different delays.

In this paper, we consider more than one implementations per module and we call this version of the clustering problem the *circuit implementation clustering* problem (CIC). Thus, we may be able to select an appropriate implementation for each module to significantly reduce the maximum delay while packaging the modules into the clusters. In the case where a module is replicated we may choose for different implementations between the replicates.

Note that the problem of selecting an implementation for each module so that the maximum delay from any primary input to any primary output is minimized, is an interesting and well studied problem by itself [1, 2, 4, 5]. However, none of the above approaches encounters that the modules are to be placed into clusters of given capacities and with significant inter-cluster delays.

In Section 2, we consider the CIC problem on combinational circuits. This appears to be a restricted problem formulation since the input is often a sequential circuit. However, in [6] it was observed that an "optimal" solution (within D units of time) on sequential circuits may be obtained by disconnecting all flip-flops at their inputs, provided an optimal clustering algorithm for the problem on the remaining combinational circuit. Thus, the CIC problem on combinational circuits is very important. We show that the decision version of the CIC problem is NP-complete in the strong sense on combinational circuits, but it is weakly NP-complete on the special case of fan-out free circuits. For the latter case, an optimal pseudo-polynomial algorithm is presented whose time complexity depends on the number of modules and the cluster capacity B.

In Section 3, we present two heuristic approaches for the CIC problem. Both heuristics apply to either combinational or sequential circuits. The first is applicable to general values of B, and the second is focusing on the special case where B is a fixed constant. The latter approach uses the optimal algorithm of Section 2.2 as a subroutine, and outperforms the former approach. Unfortunately the methodology becomes inapplicable (due to its time complexity) when B is a high order polynomial (or exponential) function of the input size. This is not often the case, however.

Both heuristics may significantly increase the overall area of the circuit since they allow for node replication. The latter is a powerful and common mechanism while clustering (see also [6, 7]) and results to significant reductions in the delay. We show how our heuristics can be modified to take into consideration a bound on the total area. To our knowledge, no heuristic has been proposed for this variation of the CIC problem. The authors of [6] only proposed two operations that reduce unnecessary node replications without increasing the delay obtained while clustering under unlimited node replications. In Section 4, we discuss our experimental results in details.

We note that the special case of the CIC problem where all the implementations of each module have uniform areas is polynomially solvable on combinational circuits. We first greedily select for each module the implementation which has minimum delay. Then we cluster the modules using the optimal algorithmin [7]. Following arguments as in [7] it is easily shown that the resulting clustering produces optimal delay. Let $w(v^j)$ denote the area of the j^{th} implementation of module v and $d(v^j)$ its delay. In the remaining of the paper, we assume

that the implementations of each module have different areas, and that any two implementations v^j and v^k of a module v either satisfy that $w(v^k) > w(v^j)$ and $d(v^k) < d(v^j)$ or that $w(v^k) < w(v^j)$ and $d(v^k) > d(v^j)$.

2. COMPLEXITY ANALYSIS OF THE CIC PROBLEM ON COMBINATIONAL AND FAN-OUT FREE CIRCUITS

2.1. Combinational Circuits

We assume that the input is a directed acyclic graph, which is a legal representation of a combinational circuit. In [2] we have shown that the following problem is NP-complete in the strong sense: *The decision version of the Basic Circuit Implementation Problem (BCI) problem.*

Instance : A combinational logic circuit with two implementations for each module m_i of the circuit, each implementation $m_i^j, 1 \leq j \leq 2$, having area $a_i^j \in Z^+$ and delay $d_i^j \in Z^+$, $A \in Z^+$ and $\Delta' \in Z^+$.

Question: Is there a circuit implementation, such that the total area of the circuit is no more than A and the maximum delay from any input to any output is no more than Δ'?

It can now be easily shown that the decision version of the CIC problem on combinational circuits, formally defined below, is NP-complete in the strong sense:

The decision version of the CIC problem on combinational circuits.

Instance: A combinational logic circuit with two implementations for each module m_i of the circuit, each implementation $m_i^j, 1 \leq j \leq 2$, having area $a_i^j \in Z^+$ and delay $d_i^j \in Z^+$ and B, D, $\Delta \in Z^+$.

Question: Is there a partitioning of the modules into clusters and an appropriate implementation of each module so that the size (in terms of area) of each cluster is at most B and the maximum delay is at most Δ, when considering a uniform inter-cluster delay D?

We emphasize that one significant difference in the definition of the BCI and CIC problems is that the latter allows for the modules to be replicated while the former does not. We add that in the CIC problem we allow for different implementations of the replicates of any module m_i.

THEOREM 2.1 *The CIC decision problem is NP-complete in the strong sense.*

Proof The CIC decision problem can be easily shown to be in NP. We show that the BCI decision problem reduces to the CIC decision problem. Given an instance of the BCI problem one simply constructs an instance of the CIC problem by setting $B = A$, $\Delta = \Delta'$ and $D = \Delta' + 1$. We now claim that the BCI instance is solvable if and only if the CIC instance is solvable on the same input combinational circuit.

It is easy to see that the CIC instance is satisfied if the BCI instance is satisfied. It remains to show that if the CIC instance is satisfied then the BCI instance is also satisfied. First observe that the assignment of values to Δ and D enforces that the CIC solution cannot have two or more clusters; if two or more clusters exist then there will be at least one input/output path along which we encounter at least one inter-cluster delay. Secondly, we show that if the CIC problem has been solved using one cluster, and some nodes have been replicated, then we can map all the replicates of each node in one node, thus reducing the area, without an increase to the maximum delay along any input/output path. This can be easily obtained by selecting for the unique node the implementation with the least delay among all the implementations of the replicates. That way, we also succeed to satisfying the BCI instance. □

Theorem 2.1 shows that we have to rely on heuristic approaches for solving the CIC problem.

2.2. An Optimal Algorithm for Fan-Out Free Circuits

In this Section, we consider the CIC problem on circuits with tree topology. Each module of the circuit can have more than one inputs but only one

output. Thus, all the circuits considered here are fan-out free circuits.

In [5] it has been shown that the BCI decision problem (as defined in the previous Section) is weakly NP-complete even for the special case of chain-graphs. We can easily reduce the BCI decision problem to the CIC decision problem on chain graphs by setting B, the capacity of the cluster size, equal to the total area A in the definition of BCI. It can be easily seen that the BCI decision problem is equivalent to the CIC decision problem by simply observing that, in the case of trees, node replication cannot be of use to clustering. Thus, unless P = NP, it is unlikely that a polynomial time algorithm can be found for the CIC problem on fan-out free circuits. However, in the following, we show that the CIC decision problem is weakly NP-complete and we present an optimal pseudo-polynomial time algorithm for the optimization version of the CIC problem on fan-out free circuits.

Let the given combinational circuit have the structure of a tree T with *root* u. Let n, m be the number of nodes in the tree and the maximum number of implementations of a module, respectively. Let $w(v^j)$ denote the weight (area) of the j^{th} implementation v^j of node $v \in T$, $1 \leq j \leq m$. We use $W(C)$ to denote the weight of cluster C, i.e., the sum of the weights on the nodes in C. Furthermore, *cluster*(v) denotes the cluster with *root* node v, i.e., the cluster whose single output node is node v.

Assuming direction from the leaf nodes to the root u, it is clear that each node $v \in T$ has one output and at least one input. Let $d(v^j)$ denote the delay of v^j, the j^{th} implementation of node v. Finally, let B, D be the cluster capacity bound and the inter-cluster delay, respectively.

As mentioned previously, in fan-out free circuits node replication is useless. We thus have to worry how to assign the nodes into clusters, and at the same time select appropriate node implementations so that the maximum delay along any path is minimized.

Our algorithm proceeds in a bottom-up manner, and *examines* every node v starting from the leaf nodes, i.e., nodes that are only fed by the primary inputs. An internal node will be examined only if all its immediate predecessors have been examined. The examination of a node v consists of the generation of m arrays $TA_v^j[1 \ldots B]$, $1 \leq j \leq m$, one array per module implementation. Then arrays $TA_v^j[1 \ldots B]$ are combined to generate array $TA_v[1 \ldots B]$ for node v.

Thus, while at node v, the algorithm consists of $m+1$ phases. The first m phases, are dedicated to constructing arrays $TA_v^j[1 \ldots B]$, the final phase for constructing array $TA_v[1 \ldots B]$. The ultimate goal of the j^{th} phase is that entry $TA_v^j[k]$ contains the minimum possible delay at the output of a cluster $C = cluster(v)$ so that $W(C) = k$ and node v has its j^{th} implementation. After all m implementations of node v have been examined, i.e., after the m^{th} phase while at node v has terminated, we proceed to constructing array $TA_v[\,]$. This array is easily constructed by setting $TA_v[k] = \min_j \{TA_v^j[k]\}$, $1 \leq j \leq m$. That way, entry $TA_v[k]$ contains the minimum possible delay at the output of a cluster $C = cluster(v)$ which has weight $W(C) = k$. Now entry $TA_v[k]$ can be used when considering later an immediate successor of node v. If v is the output module of the fan-out free circuit, the minimum delay is observed by considering $\min_i \{TA_v[i]\}$, $1 \leq i \leq B$.

Let us first describe how the j^{th} phase works while at node v. Here, we assume the j^{th} implementation of v. Independently of whether v is a leaf or internal node, the phase starts by initializing $TA_v^j[w(v^j)] = d(v^j)$ and all other entries to ∞. If v is a leaf node this completes the phase. Otherwise, the phase keeps updating $TA_v^j[\,]$ by consulting the entries of arrays $TA_u[\,]$, for all immediate predecessors u of v.

We proceed to showing how we can update entry $TA_v^j[k]$ by considering arrays $TA_v^j[\,]$ and $TA_u[\,]$ for every immediate predecessor u of node v. For each entry $TA_v^j[r]$ with $r < w(v^j)$ we set $TA_v^j[r] = \infty$. (Clearly, there isn't any *cluster*(v^j) with cluster size $W(cluster(v^j)) < w(v^j)$.

The problem of obtaining an entry $TA_v^j[k]$, $w(v^j) \leq k \leq B$, is rephrased as follows: Given a

cluster size k, find a cluster $C = cluster(v^j)$ such that the delay at the output of cluster $C = cluster(v^j)$ is minimum and $W(C) = k$. In order to obtain the minimum delay, we first calculate the maximum possible delay l at the output of v. Let u be an immediate predecessor of v and $1 \le s \le B$. We have that $l = \max_u\{\max_s\{TA_u[s] + D + d(v^j)\}\}$. Then we perform binary search on value l to find the smallest possible l such that there exists a cluster $C = cluster(v^j)$ that has weight $W(C) = k$. Note here, that every immediate predecessor u of node v which satisfies that $\min_s\{TA_u[s] + D + d(v^j)\} \le l$, $1 \le s \le B$, does not have to be clustered together with node v. Let V^p be the set of the remaining immediate predecessors of node v. Thus, while trying to obtain $cluster(v^j)$ with delay at most l, we ignore all such nodes u. Observe that our problem has been reduced to the following: Given arrays $TA_v^j[]$ and $TA_u[]$ $u \in V^p$, and l, $k \in Z^+$, is there a way of combining the entries of $TA_u[]$, $u \in V^p$ and $TA_v^j[]$ so that entry $TA_v^j[k]$ is satisfied? In other words, is there a clustering of the nodes $u \in V^p$ together with v^j in a $C = cluster(v^j)$ with $W(C) = k$ so that the delay at the single output of C is no more than l?

Let T_v be the subtree with root v having its j^{th} implementation. We first construct list L_0, which corresponds to node v. L_0 consists of only one tuple (a_1, d_1), where $a_1 = w(v^j)$ and $d_1 = TA_v^j[w(v^j)]$. Note, that entry $w(v^j)$ is the only entry of array $TA_v^j[]$ with delay different than ∞. Let the nodes in V^p be labeled as u_i, $1 \le i \le |V^p|$. Next, we consider node u_1 and we construct a list $L_{u_1}^r$ for each entry r, $1 \le r \le B$, of array $TA_{u_1}[]$. List $L_{u_1}^r$ is generated by considering entry $TA_{u_1}[r]$ with tuple (a_1, d_1) of list L_0. After we have obtained all $L_{u_1}^r$ lists, we merge them to a final list which we call L_{u_1}.

We procceed in this manner until all predecessors $u_i \in V^p$ of v have been evaluated. In general, if we want to obtain a list $L_{u_i}^r$, we consider entry $TA_{u_i}[r]$ with all tuples of list $L_{u_{i-1}}$. Furthermore, we merge all $L_{u_i}^r$ lists to obtain list L_{u_i}.

Note, that each tuple (a_i, d_i) of any list, L, must satisfy that (a) $a_i \le k$ and $d_i \le l$, and (b) L cannot contain another tuple (a_j, d_j) such that $a_i = a_j$ and $d_i \ge d_j$. However, since our goal is to find a $cluster(v^j)$ that has weight *exactly* k, it is legal for a list l to have two tuples (a_i, d_i), (a_j, d_j) such that $a_i < a_j$ and $d_i \le d_j$. If in the final list $L_{u_{vp}}$ there exists a tuple (a_q, d_q) such that $a_q = k$ and $d_q \le l$, then there is a cluster $C = cluster(v^j)$ that has $W(C) = k$ and the delay at the output of C is at most l.

Below, we give procedure $Mindelay(T_v, TA_v^j[], v^j, k, l)$

Algorithm $Mindelay(T_v, TA_v^j[], v^j, k, l)$
begin
$L_0 = \{(w(v^j), TA_v^j[w(v^j)])\}$;
if all children of v are ignored
  $L_0 = \{(w(v^j), l)\}$;
for $i = 1$ to the number of children of v **do**
  for $r = 1$ to B in array TA_i [] **do**
    $L_i^r = 0$;
    for $q = 1$ to $|L_{i-1}|$ **do**
      if $((r + a_q) \le k)$ and $((\max\{(TA_i[r] + d(v^j)), d_q\} \le l)$
      $((a_q, d_q)$ is the q^{th} element of list $L_{i-1})$;
        Append $((r + a_q), (\max\{(TA_i[r] + d(v^j)), d_q\}))$ to L_i^r;
      endif
  Obtain L_i by merging all L_i^r lists;
end

Since each array TA_{u_i} has B entries, the corresponding list L_{u_i} can have up to B tuples. Now, in order to construct a list $L_{u_i}^r$, we consider entry $TA_{u_i}[r]$ and list $L_{u_{i-1}}$. Thus, it takes $O(B)$ time to construct a list $L_{u_i}^r$. Merging all $L_{u_i}^r$ lists to obtain list L_{u_i} takes $O(B^2)$ time. Thus, the complexity of the above procedure is $O(nB^2)$.

The complete algorithm is summarized below:

Algorithm *TreeClustering*
begin
for each node v such that v is a leaf node **do**
  for each entry k of array $TA_v[]$ **do**
    if $k = w(v^j)$ for some implementation v^j
    $TA_v[k] = d(v^j)$;
  else
    $TA_v[k] = \infty$;

endif
for each node v such that v is not a leaf node **do**
(Select node v based on Topological Order);
for each implementation v^j **do**
for each k, $k < w(v^j)$ **do**
$TA_v^j[k] = \infty$;
for each k, $w(v^j) \leq k \leq B$ **do**
$l = \max_u\{\max_s\{TA_u[s] + D + d(v^j)\}\}$;
Do binary search on l;
Ignore all u, $\min_s\{TA_u[s] + D + d(v^j)\} \leq l$;
Call M *indelay*$(T_v, TA_v^j[], v^j, k, l)$;
end

Observe that the algorithm, as described here, only returns the optimal delay and not the clustering which results to that delay. However, the algorithm can be easily modified to tackle the latter information. We modify each array so that it is an array of records. Each record has two fields. The first field contains the minimum delay as before, and the second field is a pointer to a list of clusters that form a lustering, i.e., the first field of entry $TA_v^j[k]$ contains the minimum delay at the output of *cluster*(v^j) that has weight k, while the second field points to *cluster*(v^j) as well as to all clusters that contain predecessors of node v.

The example in Figure 1 illustrates our algorithm. Let the cluster size $B = 5$ and the intercluster delay $D = 20$. Furthermore, assume two implementations per module. Figure 1a gives the graph representation of the circuit under evaluation. Figure 1b gives the area and delay of the different implementations of the nodes. Figures 1c, 1d are the complete arrays for the leaf nodes v_1, v_2, respectively. Figures 1e, 1f are the arrays for node v_0, based on the nodes first and second implementation, respectively. Finally, Figure 1g gives the final array of node v_0. Next, we show how to obtain entry $TA_{v_0}^1[5]$ in Figure 1e, i.e., we show how to obtain *cluster*(v_0), that has weight W(*cluster*$(v_0)) = 5$ and the delay at the output of *cluster*(v_0) is 13, based on node v_0's first implementation, v_0^1.

$L_0 = \{(1, 6)\}$
$L_1^1 = \{(2, 14)\}$

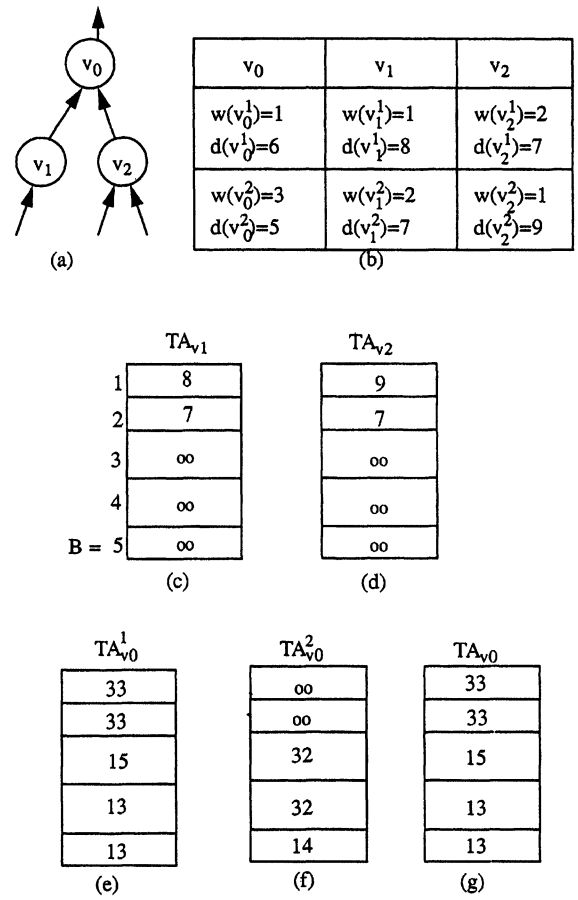

FIGURE 1 An illustration of the Optimal Algorithm for Fan-Out Free Circuits. (a) Graph representation of the Circuit under evaluation. (b) Area and Delay of the different implementations of the nodes. (c) The final array for the leaf node v_1. (d) The final array for the leaf node v_2. (e) The complete array for node v_0, based on its first implementation. (f) The complete array for node v_0, based on its second implementation. (g) The final array for node v_0.

$L_1^2 = \{(3, 13)\}$
$L_1 = \{(2, 14), (3, 13)\}$
$L_2^1 = \{(3, 15), (4, 15)\}$
$L_2^2 = \{(4, 13), (5, 13)\}$
$L_2 = \{(3, 15), (4, 13), (5, 13)\}$.

LEMMA 2.1 *Given any combinational fan-out free circuit, an area A, and a delay D, Mindelay obtains an implementation, such that the area of the circuit is exactly A, and the delay is at most D.*

Proof Every time that Mindelay constructs a list of pairs (a_j, d_j), it keeps only the pairs where $a_j \leq A$, $d_j \leq D$, and for any other pair (a_i, d_i) of the same list, if $a_i = a_j$ then $d_j < d_i$. Thus, if after the last list has been evaluated, there exists a pair (a_k, d_k) where $a_k = A$ and $d_k \leq D$, it is clear that Mindelay has obtained an implementation. □

THEOREM 2.2 *Given any combinational fan out free circuit, TreeClustering computes the optimal clustering.*

Proof We need to show, that for any node $v \in T$ and for any cluster size k, $1 \leq k \leq B$, our algorithm obtains an optimal clustering.

Suppose we want to find a cluster $C = cluster(v)$ with a minimum delay l at the output of $cluster(v)$, for some implementation v^j of v, and a cluster size of exactly k units. Clearly, an upper bound on the delay at the output of $cluster(v^j)$ is $l = \max_u\{\max_s \{TA_u[s] + D + d(v^j)\}\}$, $1 \leq s \leq B$ and u is an immediate predecessor of v. Therefore, by performing binary search on l, TreeClustering can obtain the smallest possible delay l. In addition, given a value l, if $\min_s \{TA_u[s] + D + d(v^j)\} \leq l$, it means that predecessor u can be in a different cluster than node v. However, the rest of v's predecessors have to be placed in the same cluster with v. Otherwise, the delay will be bigger than the desired l. We have pointed out that our problem has been reduced to following: Given a subtree T_v with root v^j a cluster size k, and a delay l, find $C = cluster(v^j)$ such that $W(C) = k$ and the delay at the output of C is at most l. Lemma 2.1 shows that this problem can be optimally solved. Thus, for any possible cluster size, of any implementation of each node $v \in T$, TreeClustering obtains an optimal clustering.

Now the optimality of TreeClustering is based on the following observations: First, note that for a given value of $W(cluster(v^j))$ there may be more than one possible clusterings. Different clusterings may have different delays. However, it suffices to keep only one of the clusterings that guarantee minimum delay at the output of the cluster. Secondly, while trying to obtain $C = \text{cluster}(v^j)$, with $W(C) = k$ and delay l, TreeClustering considers only predecessors $u \in V^p$. One may argue that, by considering the remaining predecessors, the adversary may successfully result to a cluster C with $W(C) = k$ and minimum delay l while TreeClustering does not. However, TreeClustering will obtain a cluster C' such that $W(C') = k'$, $k' < k$, and with minimum delay l. This is obviously an optimal step, with respect to further clusterings, i.e., when we later consider the successor of v.

For any node $v \in T$, for any cluster size k, $1 \leq k \leq B$, and provided that all immediate predecessors of node v have been evaluated, our algorithm obtains an optimal clustering. Theorem 2.2 follows. □

The time complexity of the entire algorithm can be shown to be $O(n^2\, m\, B^3 \log l)$. Assuming that m, B are constant numbers, our algorithm runs in $O(n^2 \log l)$ time.

3. HEURISTIC APPROACHES FOR COMBINATIONAL CIRCUITS

As mentioned in the Introduction, the CIC problem on a sequential circuits is reduced to a CIC problem on a combinational circuit. For this reason, our heuristics are given on combinational circuits modeled by directed acyclic graphs. For simplicity, we assume two implementations per module. The conventional approach for solving the CIC problem, first obtains an implementation for each circuit module without considering clustering and then in a second step it performs clustering based on those implementations. Our proposed heuristics obtain the implementations while performing clustering, and our experimental results show that they outperform the conventional approach.

Our first heuristic (Section 3.1) follows the *iterative improvement* methodology. Its performance is based on local issues, i.e., in order to obtain an implementation for some module it assumes fixed implementations for the rest of the modules. In Section 3.1, we describe in detail how

to obtain the implementation of each module while considering the clustering problem.

In order to overcome the local issues involved in the first heuristic, we deviced a second heuristic (Section 3.2), which uses as a subroutine the dynamic programming scheme of Section 2.2 and obtains the module implementations by considering the whole circuit. However, its complexity depends on the value of the cluster size B and thus the approach is recommended if B is a fixed constant. In this case, the second approach outperforms the former both in time response and quality of results.

Both heuristics assume unlimited node replications. This is a common methodology in clustering applications where the primary objective function is to obtain minimum delay [6, 7]. In case where an upper bound on the total area is imposed, we modify both heuristics (Section 3.3) so that we do not violate the area upper bound.

3.1. A First Approach

Our first approach uses iterative improvement. Before we describe the approach, we must define the gain $g(u)$ of a node u in the circuit. The latter is the difference between the delay on the longest path of the clustering considering the node's current and complimentary implementation, respectively. The heuristic works in phases. The major operation of a phase is to select the module with the biggest gain, change its implementation, and lock it. When a module gets locked it means that (a) its implementation has been determined, and (b) it will not be changed in later iterations within the current phase of the heuristic. Next, we describe the implementation of a phase of the heuristic in detail.

We choose randomly an initial implementation for each module and we perform clustering using the algorithm in [7]. Let the delay on the longest path of this clustering be D_1. Let M_1 be some module whose gain is to be evaluated. We perform clustering once more by considering this time the second implementation of module M_1. Let the delay on the longest path of this clustering be D_2. The gain of node M_1 is $g(M_1)=D_1-D_2$ i.e., the benefit resulting from the interchange on the module's implementation.

A phase terminates after all modules are locked. For each phase we associate a cost, namely the best delay encountered during the phase. At the end of the phase we unlock the modules, and we assign the implementations which correspond to the cost of the phase. At this point we start a new phase. We stop generating a new phase if the cost of the new phase is no better than the cost of the previous one. Although this scenario requires, in the worst case, at most D phases, where D is the maximum possible delay, we have observed that in practice the process terminates after a few phases. (In practice our heuristic is programmed to execute at most 10 phases). Thus, we only need to describe the time complexity of a phase. This is given below.

Let n be the number of modules of the given circuit and m be the number of edges that connect these modules. For every module change the heuristic calculates the gain of all unlocked modules by performing clustering using the algorithm in [7], which requires $O(n^2 \log n+nm)$ time. Therefore, the time complexity of the heuristic is $O(n^4 \log n+n^3m)$.

The latter time complexity analysis indicates that the approach is prohibitive if the circuit has more than 500 nodes. We have improved the time performance by locking only a constant number of modules at each phase. That way, the time complexity is reduced by an $\Theta(n)$ factor.

3.2. A Faster Approach for Small Values of B

Our second approach suggests to transform the circuit under evaluation to a fan-out free circuit and then use the dynamic scheme of Section 2.2 to obtain an optimal implementation for each circuit module.

The heuristic proceeds in *steps*.

Step 1: We select randomly an implementation for every module. Thus, we can assume tempora-

rily that each module has one implementation, the one chosen randomly, and perform clustering by employing the optimal algorithm for combinational circuits in [7]. Observe, that the graph produced after the clustering stage of the algorithm is also a dag. Therefore, there are modules in the graph that are not fan-out free.

Step 2: We transform the graph, produced in Step 1, to a fan-out free circuit. This is done as follows: For every output node u of the circuit we start traversing the graph following an inverse direction than the one indicated by the edges, and visiting a node after its unique successor has already been visited. That way, we obtain one disjoint circuit C_u for each output node u. For each C_u we remove all forward edges. If there are still some not fan-out free nodes, then we remove edges in a greedy manner until all modules in the graph are fan-out free.

Step 3: Now the graph is fan-out free and we are able to obtain an optimal implementation for every module by running the algorithm in Section 2.2. Note that, in Step 1 the algorithm in [7] allows for modules to be replicated. Therefore, our proposed algorithm in Section 2.2 may assign different implementations to the same module.

Step 4: We insert the edges removed at Step 2, and we perform clustering by running the algorithm in [7], on the module graph with the implementations selected at Step 3. If the maximum delay obtained at Step 4 is not larger than the maximum delay obtained prior to Step 3, we lock all modules that were assigned different implementations at Step 3. If the maximum delay obtained at Step 4 is larger than the maximum delay obtained prior to Step 3 (either at Step 1 or at Step 4 of the previous iteration, if one exists), we lock all modules that were assigned different implementations at Step 3, but the chosen implementation is the one assigned prior to Step 3. We repeat Steps 2–4 until all modules are locked.

For every unlocked module the heuristic first performs clustering by running the algorithm in [7]. This takes $O(n^2 \log n + nm)$ time. Then it obtains an implementation for each module by employing our proposed algorithm in Section 2.2. This step takes $O(n^2 k B^3 \log l)$ time, where k is the number of possible implementations per circuit module, and l is the maximum delay from any input to any output of the circuit. Therefore the heuristic requires $O(n^3 \log n + n^2 m + n^3 k B^3 \log l)$ time. However, as we mentioned above, we use this heuristic only for small fixed values of B. Moreover, we can assume that the number of possible implementations is also a constant number. Therefore, the time complexity of the heuristic reduces significantly to $O(n^3 \log n + n^3 m + n^3 \log l)$.

3.3. Considering an Area Upper Bound

The heuristics of Sections 3.1 and 3.2 allow for unlimited node replications, when clustering the nodes into sets. However, node replications often result in significant increases in the area of the clustered circuit. This is especially true when the cluster sizes are small. This Section describes a very simple approach that applies to both heuristics so that we also take into consideration an upper bound A on the total area of the circuit.

We first describe how to modify the heuristic of Section 3.1. Recall that after all phases of the heuristic have terminated, each module has been assigned an implementation. At that point, the heuristic simply calls the cluster generation procedure of the algorithm in [7] which generates the actual clusters in a bottom-up manner, starting from the outputs, and backtracking towards the inputs. (For more details, see [7]).

We modify the heuristic of Section 3.1 by controlling the cluster generation procedure, so that we take into consideration the area bound A. More specifically, each time a new cluster is generated by the clustering procedure of the algorithm in [7], our modified heuristic computes a set of nodes F that are not roots of any formed cluster, but are incident to a node in a formed cluster. (The root of a cluster in the algorithm of [7] is defined as in our algorithm of Section 2.2). The set of nodes F forms a cut, i.e., disconnects a

(not necessarily proper) subset of the primary outputs from the primary inputs. Given F, we can easily compute C_F, be the portion of the combinational circuit that expands from the primary inputs up to the cut F.

Let $A(C_F)$ denote the sum of the areas on the nodes of C_F, implemented as determined by the heuristic of Section 3.1. The idea in our modified heuristic is to cluster the nodes of C_F by applying a simple depth first search method, whenever the cluster generation procedure has already resulted in many node replications.

When a new cluster C is created by the cluster generation procedure of the algorithm in [7], we are able to determine whether the sum of the areas of the formed clusters plus $A(C_F)$ is more than the area bound A. If it is not, we proceed into generating a new cluster. Otherwise, we stop generating clusters with the procedure in [7], we consider the subcircuit C_F *prior to the generation of cluster c*, and we partition its nodes without allowing any replication. The latter is obtained by starting from the inputs of C_F and applying a depth first search traversal. Whenever a new node v of C_F is traversed is placed in the current (unfilled) cluster. When the scanned node v does not fit in the current cluster, then a new (unfilled) cluster is generated, and node v is placed in the latter cluster.

The modification of the heuristic in Section 3.2 is done along the same lines. Notice that Step 1 of the latter heuristic is in fact the algorithm in [7], where every module has a randomly chosen implementation. This step is modified as we described earlier for the heuristic of Section 3.1. That way, we do not modify the area bound A. Steps 2 and 3 need not be modified since they do not introduce any module replications. Notice that after Step 3, we have r subcircuits, where r is the number of outputs of the input circuit. At Step 4, the algorithm in [7] is applied to r different subcircuits. For each subcircuit, we apply the modification of the clustering phase in [7] described earlier, but the area upper bound is now set to be A/r.

4. EXPERIMENTAL RESULTS

We implemented our heuristics in C and run on a San Sparc System 5. For comparative reasons we also implemented the straightforward approach. Here, we assume that each module has been implemented so that the delay of the circuit is minimum, without considering clustering and without any limitations in the total area. Such an implementation is easy, as each module must be assigned the implementation that has minimum delay. Subsequently, we perform clustering using the algorithm in [7].

We experimented on several ISCAS'85 benchmarks. Since the ISCAS'85 circuits do not include a list of possible implementations for each module, we generated these implementations as suggested in [1, 5]. Namely, the first implementation of each module was assigned delay 1 and the second delay 2. The areas of the implementations were set to be inversely proportional to the delays. We also considered cluster capacities 5, 10, and 50. Every node (gate) of an ISCAS'85 circuit is treated as a separate module. The inter-cluster delay D was set to be 6. This is no more than three times the delay of a node in the circuit.

Table I gives the experimental results for the heuristics, considering cluster size 5, and unlimited node replications. This is a reasonable cluster size, up to 10 nodes are allowed per set, since most of the ISCAS'85 circuits have small depth and they do not have enough nodes (see Tab. I) so that we observe significant improvements in the delay for much larger cluster sizes. Column "Conventional" lists results for the straightforward approach, described at the beginning of the section. In particular, under "Delay" we list the maximum delay along a path from any primary input to any primary output after clustering, and under "Time" we list the total running time (time required for the implementation of the modules and doing clustering as in [7]). "Time" is expressed in seconds. Approach 1 contains results ("Delay" and "Time") for the approach of Section 3.1. Each phase of the heuristic terminates after *half* of the

TABLE I The performance of our heuristics for cluster size 5

Circuit ISCAS	# of Nodes	Depth of Circuit	Conventional		Approach 1		Approach 2	
			Delay	Time	Delay	Time	Delay	Time
C432	196	22	70	2	53	120	53	50
C499	243	24	42	3	39	235	39	115
C880	443	25	83	6	73	380	71	205
C1355	587	25	91	30	84	1950	80	1060
C1908	913	41	128	60	118	3500	113	2345
c2670	1350	33	111	85	80	7125	80	5112
c3540	1719	48	156	115	131	13425	129	8350
c6288	2448	125	377	250	343	28500	388	19835

modules are locked within a phase. Similarly, Approach 2 contains results ("Delay" and "Time") for the approach in Section 3.2. We note that the time performance results listed for both approaches do not include the actual cluster generation phase. (This is not needed in order to compute the delay). Thus, the time responses are independent of the cluster size.

Table I clearly shows that for $B=5$ both proposed heuristics outperform the conventional approach in quality of results. We observe up to 25% savings on the overall delay over the conventional approach. Thus, the proposed methodology of combining the module implementation phase with the clustering phase is very promising.

Tables II and III show similar comparative experimental results for cluster sizes 10 and 50, respectively. For cluster size 10, we allow up to 20 modules per cluster, the improvements (in the delay) over the conventional approach is decreased, but it is still significant on some benchmarks. For cluster size 50, up to 100 modules are allowed per cluster, we only observed improvements for the large ISCAS'85 benchmarks that have larger number of nodes and larger depth. It is noteworthy that for all the above values of B, the proposed approach in Section 3.2 outperforms the one in Section 3.1 in time and quality of results. This may be justified because, due to time limitations, we did not allow Approach 1 to lock more than 20 modules.

Table IV shows the effect of node duplications while clustering, with respect to minimizing the overall delay. It also shows the effectiveness of the modification of the heuristic of Section 3.1, as described in Section 3.3, so that we do not exceed an upper bound A on the overall area of the circuit. The third column (includes two subcolumns) shows the overall area of each circuit as well as the delay when we are allowed to have unlimited node replications. (Here we have applied the operations *cluster merging* and *node deletion* of

TABLE II The performance of our heuristics for cluster size 10

Circuit ISCAS	# of Nodes	Depth of Circuit	Conventional		Approach 1		Approach 2	
			Delay	Time	Delay	Time	Delay	Time
C432	196	22	42	2	40	120	40	50
C499	243	24	30	3	30	235	30	115
C880	443	25	55	6	54	380	53	205
C1355	587	25	55	30	55	1950	54	1060
C1908	913	41	91	60	86	3500	84	2345
c2670	1350	33	79	85	67	7125	67	5112
c3540	1719	48	110	115	102	13425	98	8350
c6288	2448	125	281	250	262	28500	259	19835

TABLE III The performance of our heuristics for cluster size 50

Circuit ISCAS	# of Nodes	Depth of Circuit	Conventional Delay	Conventional Time	Approach 1 Delay	Approach 1 Time	Approach 2 Delay	Approach 2 Time
c2670	1350	33	46	85	45	7125	45	5112
c3540	1719	48	66	115	62	13425	661	8350
c6288	2448	125	167	250	159	28500	159	19835

TABLE IV Area/Delay tradeoffs for cluster sizes 5 and 10

Circuit ISCAS	Cluster Size	Unlimited Repl. Area ($\mathcal{A}$)	Unlimited Repl. Delay	Init + [$\mathcal{A}/2$ − $\mathcal{A}/16$] Delay	Init + 500 Delay	Init + 0.5 Delay
C432	5	54578	61	73	73	111
C432	10	38250	51	57	63	99
C499	5	4957	43	49	49	82
C499	10	4644	33	45	45	70
C880	5	4913	79	85	96	140
C880	10	4330	59	65	75	133

[6]) that reduce the overall area without increasing the delay). The remaining columns (fourth, fifth, and sixth) show the delay obtained by the heuristic of Section 3.3 when the upper bound A in the overall area is set to be Init $+A_x$, where Init is the area of the circuit without any node duplication (i.e., before clustering), and A_x is an additional area increase that varies from column to column. For example, in the fourth column A_x is anywhere in the range [$\mathcal{A}/2 - \mathcal{A}/16$], where $\mathcal{A}$ is the overall area of the clustered circuit under unlimited node replications. Here, for all listed benchmarks, the delay was the same for all values of A_x in the abovementioned range. (More precisely, more than 50 discrete values were examined in that range, and the delays were identical for each circuit). In the fifth column $A_x = 500$, and in the sixth column it is 0.5. In the last column, the value of A_x practically does not allow for any node replication. The table shows that the modification of the heuristic of Section 3.1 (as described in Section 3.3) yields a promising technique when a reasonable area overhead is allowed. It also shows that when no node duplications are allowed, the delay may be increased, in some case, by more than 100%. In general, the observed increases in the delay indicate that clustering without node replications is unacceptable in timing driven clustering applications. Similar results where obtained when we applied

TABLE V The performance of Approach 1 for cluster sizes 5 and 10 on sequential circuits

Circuit ISCAS	# of Nodes	Cluster Size	Conventional Delay	Conventional Time	Approach 1 Delay	Approach 1 Time
s208.1	122	5	35	0.6	30	50
s208.1	122	10	24	0.6	23	50
s298	137	5	32	0.9	25	95
s298	137	10	21	0.9	18	95
s349	185	5	74	1.4	54	118
s349	185	10	42	1.4	39	118
s386	172	5	42	1.3	32	115
s386	172	10	24	1.3	22	115
s444	204	5	42	1.8	33	368
s444	204	10	24	1.8	21	368

the modified heuristic of Section 3.2, as described in Section 3.3, so that it considers an upper bound A on the total area.

Finally, Table V shows experimental results on some of the ISCAS'89 sequential circuits. These results were obtained by first breaking all flip-flops at their inputs and then applying the heuristic of Section 3.1.

5. CONCLUSION

We consider a more realistic version of the circuit clustering problem, where each module has more than one possible implementations. We call this problem the CIC problem. We consider the CIC problem on fan-out free circuits provided that different implementations may have different areas, and we give a pseudo-polynomial time algorithm. We show that the problem is NP-hard on combinational circuits, in general. Furthermore, we propose the first heuristics for the CIC problem. Our heuristics may also consider a bound on the total area of the circuit after clustering. Increases in the area are due to node replications while clustering.

Acknowledgements

Research supported by NSF grant MIP-9409905.

References

[1] Chan, P. K. (1990). "Algorithms for library-specific sizing of combinational logic", *27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 353–356.

[2] Karayiannis, D. G. and Tragoudas, S., "Uniform Area Timing-Driven General Circuit Implementation", *Proceedings of the 5th Great Lakes Symposium on VLSI*, March 16–18, Buffalo, NY, pp. 2–7.(An extended version appears as TR CS 94-04, Computer Science Department, Southern Illinois University, Carbondale, IL 62901).

[3] Lengauer, T. (1990). *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Tuebner.

[4] Li, W. N. (1993). "Strongly NP-hard Discrete Gate Sizing Problems", *Proc. IEEE Int. Conf. Computer Design (ICCD '93)*, pp. 468–471.

[5] Li, W., Lim, A., Agrawal, P. and Sahni, S., "On The Circuit Implementation Problem", *IEEE Trans. on CAD*, **12**(8), 1993, 1147–1156. A preliminary version appears in the *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '92)*, 1992, pp. 478–483.

[6] Murgai, R., Brayton, R. K. and Sangiovanni-Vincentelli, A. (1991). "On Clustering for Minimum Delay/Area", *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1991, pp. 6–9.

[7] Rajaraman, R. and Wong, D. F. (1993). "Optimal Clustering for Delay Minimization", *Proceedings of the 30th ACM/IEEE Design Automation Conference (DAC '93)*, pp. 309–314.

[8] Sherwani, N. (1993). *Algorithms for VLSI Physical Design Automation*, Kluwer.

Authors' Biographies

Dimitrios G. Karayiannis was born in Greece, on March 12, 1969. He received the B.S. degree in Computer Science from Southern Illinois University in 1991 and the M.S. degree in Computer Science from Southern Illinois University in 1993. Currently. He received the Ph.D. degree at Southern Illinois University in 1996.

From 1991 to 1993, he was a teaching assistant. From 1993 to 1996 he was a research assistant in the department of Computer Science, Southern Illinois University. Since 1997 he has been with Viewlogic Systems Inc. His research interests include Computer Aided Design (algorithms and applications), Design for Testability, Test Pattern Generation, and Built-In Self-Test.

Spyros Tragoudas received his Diploma degree in Computer Engineering from the University of Patras, Greece in July 1986 and his M.S. degree in Computer Science from the University of Texas at Dallas in August 1988. He received his Ph.D. degree in Computer Science from the University of Texas at Dallas in August 1991. He joined the faculty of Southern Illinois University at Carbondale in August 1991, where he is currently an Associate Professor of Computer Science.

His research interests include Computer Aided Design for VLSI layout, Testing, and algorithms for combinatorial optimization problems. Dr. Tragoudas is a member of IEEE Computer and CAS societies. He received the 1994 ICCD Outstanding Paper Award, Design and Test Truck.