Simultaneous Circuit Partitioning/Clustering with Retiming for Performance Optimization

Jason Cong and Honching Li and Chang Wu
Department of Computer Science
University of California, Los Angeles, CA 90095
{cong, honching, changwu}@cs.ucla.edu

Abstract

Partitioning and clustering are crucial steps in circuit layout for handling large scale designs enabled by the deep submicron technologies. Retiming is an important sequential logic optimization technique for reducing the clock period by optimally repositioning flipflops [7]. In our exploration of a logical and physical co-design flow, we developed a highly efficient algorithm on combining retiming with circuit partitioning or clustering for clock period minimization. Compared with the recent result by Pan et al. [10] on quasi-optimal clustering with retiming, our algorithm is able to reduce both runtime and memory requirement by one order of magnitude without losing quality. Our results show that our algorithm can be over 1000X faster for large designs.

1 Introduction

Retiming is a very important sequential logic optimization technique to minimize the clock period by optimally repositioning flipflops (FFs) [7]. However, it is usually performed at the logic design level and suffers from the lack of estimation of the interconnect delay, which becomes more and more important for large deep submicron designs. Circuit clustering and partitioning are two very important techniques for large scale system design and provide the first order of information about local and global interconnects. However, traditionally, retiming and partitioning or clustering are usually separated. Most existing clustering algorithms consider only combinational circuits [6, 12, 9]. For sequential circuits, they need to cut off all the flipflops and process each combinational subcircuit independently, which may only lead to inferior results. On the other hand, most partitioning algorithms minimize only the cut size between different partitions without considering the performance issue [4, 11, 5]. Recent work by Liu et al. [8] proposed to combine retiming and logic replication for bi-partitioning for clock period minimization. However, for general k-way partitioning, iterative application of such bi-partitioning often lead to sub-optimal solutions.

Recently, Pan et al. [10] proposed a polynomial time quasi-optimal clustering with retiming algorithm which guarantees to produce a clustering solution with delay of no more than D over the minimum delay, where D is the maximum interconnect delay between clusters. For circuit with n gates

and m edges, the time complexity is $O(B \cdot n \cdot m \cdot log^2 n)$, where $B = O(n^2 D)$. Although B can be very small in practice with careful implementation of the algorithm, the overall complexity is still very high for large designs. (Our test indicates that it takes more than 20 hours without generating a result for a circuit with 9817 nodes on a Sun Enterprise 4000 with 1.5GB memory.) Moreover, the algorithm needs an n by n matrix to store all-pair longest paths before computing an optimal clustering, which requires prohibitively large amount of memory for designs with over 100,000 gates.

In this paper, we consider a generic framework for both clustering and k-way partitioning with simultaneous retiming for performance optimization. We consider the general delay model and partition a sequential circuit into separate blocks under the area bound constraint. For clustering, the area bound A of each cluster is a given constant. For k-way partitioning, we set $A \approx \frac{n}{k}$.[1] We propose a highly efficient and scalable quasi-optimal algorithm for sequential circuits with retiming to minimize the clock period. Our algorithm works in $O(B \cdot A \cdot n \cdot logn)$ time, where B is a very small number (4 to 25) in practice. Furthermore, our algorithm does not need to pre-compute and store the huge all-pair longest-path matrix, and reduces the space complexity from $O(n^2 + m)$ to $O(A \cdot n + m)$ which is the limiting factor for large designs.. Although our algorithm is mainly for clustering, it can be applied for k-way partitioning as well.

The remainder of the paper is organized as follows. Section 2 presents the problem formulation and preliminaries. Section 3 presents a review of the algorithm in [10]. Our algorithm is presented in Section 4. The experimental results are presented in Section 5. Discussions and conclusions are given in Section 6. Due to page limit all proofs of the theorems are omitted and can be found in [2].

2 Problem Formulation and Preliminaries

The clustering problem is to decompose a given circuit into a number of clusters such that their sizes are bounded by a given number A. For performance optimization, we study the following problem.

Problem 1 *For a sequential circuit, construct a clustered circuit with the minimum clock period under retiming with possible node replication. The area of each cluster is bounded by a given number A.*

As in [9, 12, 10], the *general delay model* is used in this paper, which assumes that each gate v has a delay of d_v and each interconnect between clusters has a delay of D,

---

[1] This may not guarantee exact k partitions after clustering. Post-processing of merging small clusters may be needed.

the local interconnect delay within each cluster is 0.[2] The size of each gate v is a_v. We ignore the setup and hold times and the size of flipflops (FFs) as previous works on retiming [7, 10]. We assume that both D and d_v are integers.[3] The computation of minimum clock period is achieved by solving a sequence of the decision problem formulated below:

Problem 2 *For a sequential circuit with a given target clock period ϕ and given area bound on each cluster, decide if there exists a clustered circuit with retiming and possible logic replication with a clock period of no more than ϕ under the general delay model.*

We use $G(V, E, W)$ or G to denote the *retiming graph* [7] of a sequential circuit, where V is the set of nodes representing gates in the circuit, E is the set of edges representing the connections between gates, W is the set of edge weights. The number of nodes in G is denoted n and the number of edges is denoted m. Edge $e(u, v)$ denotes the connection from gate u to gate v and $w(e)$ denotes the number of FFs on the connection. The *delay* of a node v is d_v. For an edge $e(u, v)$ and a given clock period ϕ, the *edge length*, denoted $length(e)$, is defined to be $-\phi \cdot w(e) + d_v$. The *path length*, denoted $length(p)$ of a path p, is $\sum_{e \in p} length(e)$. Intuitively, the length of a path represents the node delay on the path less the delay which can be reduced with retiming. More specifically, in a clustered circuit C, the delay of an edge $e(u, v)$, denoted $d_C(e)$, is 0 if u and v are in the same cluster, and D if they are in different clusters. The *length* of an edge e in C, denoted $length_C(e)$, is $length(e) + d_C(e)$. The path *length* of a path p in C, denoted $length_C(p)$, is $\sum_{e \in p} length_C(e)$. The *l-value* $l_C(v)$ of a node v in C is the maximum path length from primary inputs (PIs) to v in C. Based on the retiming theory, it was shown in [10] that:

Theorem 1 *In a clustered circuit C of a sequential circuit with a target clock period ϕ, if there is a primary output (PO) whose l-value is greater than ϕ, the clustered circuit C cannot be retimed to ϕ or less. If, on the other hand, the l-values of all POs are less than or equal to ϕ, C can be retimed to a clock period less than $\phi + D$.*

Basically, the clock period computed with l-values will differ with the minimum clock period no more than D. Under a target clock period ϕ, for every node v in the original circuit, let node label, denoted $l^{opt}(v)$, be the *minimum* $l_C(v)$ among *all clustered circuits* C. Based on Theorem 1, to check if there exists a feasible solution for a given ϕ, one can compute node labels and check if $l^{opt}(PO) \leq \phi$ holds for every PO. Let ϕ_{min} be the minimum clock period computed by labeling, $\hat{\phi}$ be the achieved clock period by labeling, and ϕ^* be the real minimum clock period among all clustered circuits. We have that:

Corollary 1 $\phi_{min} \leq \phi^* \leq \hat{\phi} < \phi_{min} + D.$

Therefore, computing ϕ_{min} with label computation, we can guarantee to find a clustered circuit with clock period $\hat{\phi}$ less than $\phi_{min} + D \leq \phi^* + D$.

[2] Local interconnect delay can be estimated and the average number can be lumped into the node delay d_v for simplicity.

[3] In case D and d_v are real numbers, we can scale them based on the required precision on clock period estimation. For example, the typical values of D and d_v in current technologies is $d_v = 50ps \sim 100ps$, $D = 200 \sim 500ps$ (for global interconnect of length $5 \sim 20$ mm) after proper buffer insertion and wire sizing. We can scale them to be $d_v = 1$ and $D = 2 \sim 10$.

3 Review of the Previous Work

In this section we review the labeling algorithm in [10] for checking the feasibility for a target clock period. In [10], the authors propose to solve a *simple clustering problem* for a target clock period. A clustered circuit C is *simple* if:

1) Each cluster has only one output, which is called the *root* of the cluster;

2) For each node, there is at most one cluster in C rooted at the node;

3) If the cluster rooted at u is connected to the cluster rooted at v, the cluster rooted at v must not contain a copy of u.

Note that, although in general each cluster can have multiple outputs, the simple clustering problem considers clusters with only one output. It was shown in [10] that there exists a simple clustered circuit whose clock period after retiming is the *minimum* clock period among all clustered circuits with retiming. The second and third criterion are mainly for restricting unnecessary node duplications which cannot help to reduce the clock period.

The major step in checking the feasibility of a target clock period is the label computation. It starts with lower bounds $l(v)$ on $l^{opt}(v)$ and repeatedly increases their values until they all converge to $l^{opt}(v)$. The initial lower bounds are zero for PIs and $-\infty$ for the other nodes. Based on the current set of $l(v)$, a tighter lower bound $l_{new}(v)$ is computed in each step as follows.

Let $\Delta(u, v)$ be the maximum path length from u to v in the original circuit. Based on the current set of lower-bounds $l(u)$, the *height* of a cluster C_v rooted at v is defined to be $h(C_v) = max\{l(u) + \Delta(u, v) + D \mid \forall u \text{ is an input to } C_v\}$. A cluster C_v is *legal* if $\mid C_v \mid \leq A$. The new lower bound of v is

$$l_{new}(v) = \min_{v \text{ legal } C_v \text{ rooted at } v} \{h(C_v)\}. \qquad (1)$$

It was computed with binary search of values in the set of $\{l(u) + \Delta(u, v) + D \mid \forall v \in V\}$.

For each target value L for $l_{new}(v)$ in binary search, it was proposed in [10] to construct a (minimum volume) cluster $C_{v,L}$ and check if its size is larger than A, where $C_{v,L} = \{u \mid u \rightsquigarrow v \text{ and } l(u) + \Delta(u, v) + D > L\}$. Clearly, $l_{new}(v) \leq L$ if $\mid C_{v,L} \mid \leq A$. Otherwise, $l_{new}(v) > L$. $C_{v,L}$ can be computed in $O(n + m)$ time if $\Delta(u, v)$ is known for every u. Since there are at most n candidates, $l_{new}(v)$ can be computed in $O((n+m)logn)$ time if the $\Delta(u, v)$ for every u is known. Let one *labeling iteration* be the process that each node's label is updated once; B be the number of labeling iterations needed to compute all node labels for each target clock period ϕ. The labeling time for a given clock period ϕ is thus $O(Bn(n + m)logn)$. For a given ϕ, $\Delta(u, v)$ can be pre-computed and kept in an n by n matrix. The computation time for all $\Delta(u, v)$ is $O(n^2 logn + nm)$ [3]. Binary searching from 1 to nD, the minimum clock period can then be computed in $O(Bn(n + m)log^2 n)$ time with space requirement of $O(n^2 + m)$, where $B = O(n^2 D)$. Clearly, the time complexity is high for large designs with over 100,000 gates. Most of all, the quadratic order of space requirement is impractical for large applications.

4 Optimal Label Computation

In this section, we propose a highly efficient and scalable label computation algorithm to compute quasi-optimal clustering with retiming for clock period minimization. Our algorithm can be used for both clustering (A is fixed) and

461

```
Label(G(V,E,W),φ,A)
1    set initial lower-bounds of all nodes
     sort all nodes based on depth-first search (DFS) from
     PIs to POs
2    for i from 1 to B
3        set converge = TRUE
4        for each node v in the DFS order
5            l'_new(v) = LabelUpdate(v,φ,A)
6            if (l'_new(v) > l(v))
7                set converge = FALSE and l(v) = l'_new(v)
8                if l(v) > φ for a PO v, return(FALSE)
9        if (converge == TRUE), return(TRUE)
10   return(FALSE)
```

Figure 1: Label computation for a target clock period ϕ and area bound A on each cluster.

k-way partitioning ($A \approx \frac{n}{k}$). It goes through the following steps:

1. Binary search to get the minimum clock period with label computation,

2. Form the clustered circuit based on node labels with respect to the minimum clock period computed,

3. Retiming to achieve the best clock period.

Notice that, although we separate retiming with clustering in different steps, our label computation in Step 1 guarantees that the clock period computed is bounded by $\phi^* + D$ according to Corollary 1. In the next subsection, we will present a much faster label computation procedure for achieving the minimum clock period, which is our major contribution to solving the clustering problem very efficiently.

4.1 Iterative Label Computation for the Minimum Clock Period

We binary search a range [lb,up] to compute the minimum clock period ϕ^*, where lb and up are a lower-bound and an upper-bound on the value of the minimum clock period precomputed by existing algorithms. For each target clock period ϕ, we compute the node labels with respect to ϕ and check if $l^{opt}(PO) \leq \phi$. If $l^{opt}(v) \leq \phi$ hold for all the POs v, we guarantee that $\phi^* < \phi + D$ and start to check another smaller target clock period. If, however, there exists one PO v with $l^{opt}(v) > \phi$, we claim that $\phi^* > \phi$ and increase the target clock period for next round of checking.

To compute all node labels $l^{opt}(v)$ for a given ϕ, we assign a lower-bound $l(v)$ on the value of $l^{opt}(v)$ and iteratively update them until they all converge to $l^{opt}(v)$, or if we can determine that ϕ is not a feasible value. Initially, $l(v) = 0$ for PIs v and $l(v) = -\infty$ for all the other nodes v. A pseudo code of the label computation is shown in Figure 1, where $B = n(n-1)D$ is an upper-bound on the number of labeling iterations to guarantee that all $l(v)$ will converge to $l^{opt}(v)$. In practice, by labeling nodes in the depth-first search order from PIs to POs, our algorithm can always terminate in only a few iterations (ranging from 4 to 20 for all the examples we tested with 100 to 100,000 gates).

In the remainder of this subsection, we present our efficient label update procedure LabelUpdate(v, ϕ, A) for every node v with target clock period ϕ and given area bound A.

4.1.1 Monotone Property for Label Update in Reduced Candidate Set

To compute $l_{new}(v)$ defined in Eqn. 1 for a node based on the current set of lower bounds $l(u)$, the method in [10] performs binary search among all $O(n)$ candidates in $\{l(u) + \Delta(u,v) + D \mid \forall\, v \in V\}$. In our algorithm, however, we compute a tighter lower bound $l'_{new}(v)$ by binary searching only $D + 1$ candidates, which speeds up the algorithm by a factor of $O(logn)$. More precisely, let

$$\mathcal{L}(v) = \max_{\forall e(u,v) \in E} \{l(u) + length(e(u,v))\}.$$

We compute a tighter lower-bound $l'_{new}(v)$ by searching only the $D + 1$ integer values in the range of $[\mathcal{L}(v), \mathcal{L}(v) + D]$ as:

$$l'_{new}(v) = \min\{h(C_v) \mid \mathcal{L}(v) \leq h(C_v) \leq \mathcal{L}(v) + D\}, \quad (2)$$

where each C_v is a legal cluster rooted at v with area bounded by A. The correctness of our approach is based on the monotone property of node labels.

We say a set of $\{l(v)\}$ with respect to a given ϕ is monotone if for any edge $e(u,v)$, $l(u) + length(e) \leq l(v)$, where $length(e) = -\phi \cdot w(e) + d_v$. It represents the monotonically increasing nature of node labels from PIs to POs.

Theorem 2 (Monotone Property) *If a sequential circuit has a clustered circuit with the clock period of no more than a given ϕ under retiming, the node labels $l^{opt}(v)$ with respect to ϕ are monotone. That is, $l^{opt}(u) + length(u,v) \leq l^{opt}(v)$ for every edge $e(u,v)$ in the retiming graph of the circuit.*

Theorem 3 *If a sequential circuit has a clustered circuit with the clock period of no more than a given ϕ under retiming, the inequalities $l(v) \leq l'_{new}(v) \leq l^{opt}(v)$ hold after any sequence of label updates, which $l'_{new}(v)$ is computed according to Eqn. 2.*

Intuitively, the node labels monotonically increase for nodes from PIs to POs; and the lower-bounds we computed will also monotonically increase and converge to the node labels.

4.1.2 Longest Path Computation in Linear Time

For a node v to check if there exists a valid cluster $C_{v,L}$ with size of no more than A for a target value L, the authors of [10] propose to precompute and store all-pair longest pathes $\Delta(u,v)$ in an n by n matrix. This is very inefficient for large designs. First, to precompute the all-pair longest path, one needs $O(n^2 logn + nm)$ time with $O(n^2 + m)$ space [3]. Second, for each label update candidate value L, it needs $O(n + m)$ time to search every node u which has a path to v. In particular, the $O(n^2 + m)$ space requirement is a serious limitation which prevents the algorithm to scale to large designs (with over 100,000 gates). In the following we shall show that we can check the feasibility of each L in at most $O(K \cdot A \cdot D)$ time with $O(A)$ space, where K is maximum number of fanins of each node. In case A is large, we shall propose two speedup techniques in the next subsection.

Based on the monotone property of node labels, we propose to construct $C_{v,L}$ and compute $\Delta(u,v)$ simultaneously. Our computation is so efficient that we can afford to compute $\Delta(u,v)$ many times without precomputation and large storage space.

Since $length(e(u,v)) = -\phi \cdot w(e) + d_v$ can be either negative or positive, in general, to compute all the $\Delta(u,v)$ with

462

respect to a given v one needs to perform the well-known Bellman-Ford algorithm which takes $O(m \cdot n)$ time for a graph with n nodes and m edges [3]. However, we are able to reduce the time complexity to $O(A \cdot \log D)$ by making use of the monotone property of node labels. This is our major contribution to the performance-driven clustering with retiming problem, as it reduces one order of magnitude of both runtime and space requirement of the previous work in [10] for A to be a constant.

Our method is similar to but even faster than Dijkstra's shortest path algorithm, which applies to special graphs with only positive edge length [3]. Even for this kind of special graphs, Dijkstra's algorithm needs $O(n \cdot \log n + m)$ time [3].

For a given node v and target lower-bound L on v's node label, we simultaneously compute $\Delta(u,v)$ and construct $C_{v,L}$ and check if its size is no more than A. The detailed procedure is as follows: We start from the root v and grow $C_{v,L}$ progressively towards PIs. At the beginning, $C_{v,L} = \emptyset$. To compute $\Delta(u,v)$, we assign a lower-bound $\delta_l(u,v)$ on $\Delta(u,v)$ which is $-\infty$ for each u, and 0 for v. In each step, we pick up one node u with the maximum $l(u) + \delta_l(u,v)$ from nodes outside the current $C_{v,L}$. If $l(u) + \delta_l(u,v) + D \leq L$ for the node u and the size of the current $C_{v,L}$ is no more than A, we claim that L is a feasible value, i.e., $l'_{new}(v) \leq L$. If, however, $l(u) + \delta_l(u,v) + D > L$, we put u in $C_{v,L}$ and check if $\mid C_{v,L} \mid > A$. In case $\mid C_{v,L} \mid > A$, we claim L is an infeasible value, i.e., $l'_{new}(v) > L$. In case $\mid C_{v,L} \mid \leq A$, we update $\delta_l(w)$ as following for all the fanins w of u and continue the process by picking up another outside node u' with the maximum $l(u') + \delta_l(u',v)$. For every fanin w of u outside the current $C_{v,L}$, we update $\delta_l(w,v)$ to be $\delta_l(u,v) + length(e(w,u))$ if $\delta_l(w,v) < \delta_l(u,v) + length(e(w,u))$. Otherwise keep $\delta_l(w,v)$ to be unchanged.

The remaining problem is how to select an outside node u with the maximum $l(u) + \delta_l(u,v)$ to grow $C_{v,L}$ in each step. Since our interest is on deciding if $l(u) + \delta_l(u,v) + D \leq L$ or not, we only consider those nodes u with $l(u) + \delta_l(u,v)$ in the range of $[L - D, L]$. The reason in doing in this way is that nodes with $l(u) + \delta_l(u,v) < L - D$ will not be in $C_{v,L}$ and need not to be considered; nodes with the value larger than L can be treated the same as those with the value of L based on the monotone property. This enables a linear time bucket sort for outside nodes u. Clearly, $D + 1$ buckets are enough. To select one node u with the maximum $l(u) + \delta_l(u,v)$ from the buckets needs only $O(\log D)$ time. To update $\delta_l(w,v)$ for the fanins w of u takes $O(K)$ time, where K is the maximum fanin number of each node in the circuit. As we only need to pick A nodes to construct $C_{v,L}$, the total computation time for checking one L is $O(K \cdot A \cdot \log D)$.

One major difference between our algorithm with the Dijkstra's algorithm is how to pick up a node u to grow $C_{v,L}$. Dijkstra's algorithm picks up one with the maximum $\delta_l(u,v)$ which guarantees to be $\Delta(u,v)$ in case all edge length are negative.[4] Our algorithm, however, picks up a node u with the maximum $l(u) + \delta_l(u,v)$ in the range of $[L - D, L]$. Within this range, we can easily pick a node u with the maximum $l(u) + \delta_l(u,v)$ with simple and fast bucket sort. The correctness of our approach is based on the following theorem.

Theorem 4 *If the current set of $l(u)$ satisfies the monotone property, i.e., $l(v) \geq l(u) + length(e(u,v))$ for any edge*

---

[4]Notice that we compute the longest path, instead of the shortest path here.

$e(u,v)$, $\delta_l(u,v)$ *computed by our algorithm equals to* $\Delta(u,v)$.

During our labeling process, however, the current set of $\{l(u)\}$ may not always satisfy the monotone property, thus, the $\delta_l(u,v)$ we computed may be less than $\Delta(u,v)$. However, in this case, our label computation process will not terminate because there must exist one edge $e(u,v)$ such that $l(u) + length(e(u,v)) > l(v)$. No matter whether we compute the correct $\Delta(u,v)$ or not, the following inequalities still holds: $l'_{new}(v) \geq \mathcal{L}(v) \geq l(u) + length(e(u,v)) > l(v)$. As a result, our labeling process will continue until the current set of $\{l(u)\}$ satisfies the monotone property and then, we guarantee to compute the correct $\Delta(u,v)$. Consequently, we guarantee to compute the correct node labels when our label computation terminates.

4.1.3 Further Speedup of the Label Computation

For constant A, our algorithm runs in linear time. However, when $A = O(n)$, for example, in the case of k-way partitioning with $A \approx \frac{n}{k}$, the algorithm still runs in quadratic order, which is impractical for large designs with over one million gates. In the following, we propose two methods to quickly predict if $\mid C_{v,L} \mid \leq A$ without really constructing $C_{v,L}$ to speed up the label update procedure.

For each node v, we keep track of the size of $C_{v,l(v)}$ constructed in the previous step and save the value as $area(C_v)$ for each v. Initially, $area(C_v) = a_v$, which means that each v itself is a cluster. For a node v and a target L, let a (direct) fanin u be *critical* if $l(u) + length(e(u,v)) + D > L$, which means $u \in C_{v,L}$. We compute an upper-bound $\mathcal{U}(v,L)$ on $\mid C_{v,L} \mid$ as $\mathcal{U}(v,L) = \sum \{area(C_u) \mid \forall critical\ fanin\ u\ of\ v\}$. Obviously, if $\mathcal{U}(v,L) \leq A$, $\mid C_{v,L} \mid \leq \mathcal{U}(v,L) \leq A$ and we can conclude that L is a feasible value, i.e., $l'_{new}(v) \leq L$. If, however, $\mathcal{U}(v,L) > A$, we explicitly construct $C_{v,L}$ and check if its size is no larger than A. After getting $l'_{new}(v)$, we assign $area(C_v)$ to be $\mathcal{U}(v,L)$ in case $\mathcal{U}(v,L) \leq A$, or $\mid C_{v,l'_{new}(v)} \mid$ in case $\mathcal{U}(v,L) > A$. For larger A, the probability that $\mathcal{U}(v,L) \leq A$ is higher, thus, the runtime can be reduced without really constructing $C_{v,L}$ in many cases.

Our second method on reducing the runtime is based on the concept of *stable* and *active* nodes. A node v is *stable* if we can conclude that $l'_{new}(v) = l^{opt}(v)$ and we do not need to update the label of v any more. Otherwise, v is *active*. The details of our method are as follows.

Initially, all nodes are active, except PIs which are stable. For the current node v during the labeling process, if v is active, we compute $l'_{new}(v)$ as usual by checking the values in $[\mathcal{L}(v), \mathcal{L}(v) + D]$. Suppose $C_{v,l'_{new}(v)}$ is a cluster constructed for $l'_{new}(v)$. We mark v as a stable node if all inputs to the cluster are stable. Otherwise, we mark v as an active node. If the current v is stable, we will *not* update $l(v)$ if the monotone property satisfied on every fanin edge of v. In other words, if $l(v) \geq \mathcal{L}(v)$ and v is stable, we keep $l(v)$ unchanged.

To further explain the label update based on idea of active/stable nodes, let us consider an extreme case of a loopless circuit. We label all the nodes in a topological order from PIs to POs. Initially, all PIs are stable. The direct fanouts of PIs will also be marked stable after we update the labels for them, since their cluster inputs must all be PIs which are stable. By mathematical induction, when we reach a node v along the topologic order, all its predecessors are marked stable. As a result v will also be marked stable immediately after we update its label. After every node label being updated once, all nodes will be marked

stable with $l(v) = l^{opt}(v)$, and the label computation can be stopped. For sequential circuits with feedback loops, a few more iterations may be needed.

We can prove by mathematical induction that:

Theorem 5 *Suppose initially only PIs are marked stable. Let $l'_{new}(v)$ be an updated lower-bound on $l^{opt}(v)$ just computed. If all inputs to cluster $C_{v,l'_{new}(v)}$ are stable, then $l'_{new}(v) = l^{opt}(v)$, and v is also stable.*

Notice however, our algorithm may compute a smaller cluster $c_{v,L} \subseteq C_{v,L}$ in case the current set of $\{l(u)\}$ does not satisfy the monotone property, thus, $l'_{new}(v)$ may still be less than $l^{opt}(v)$ even if all inputs to $c_{v,L}$ are stable. To guarantee the optimality of our algorithm, if after one iteration no $l(v)$ has been changed, we perform another labeling iteration as normal without the active/stable strategy, i.e., we reset all nodes except PIs to be active again and perform another iteration of label update. If then, there are some nodes v whose $l(v)$ have been increased, we continue the labeling process by enabling the active/stable strategy again. Otherwise, if there is no node whose $l(v)$ has been increased, we can safely conclude that $l(v) = l^{opt}(v)$ for every node v and the labeling process can be terminated. Our experimental results show that for $A = \frac{n}{16}$, these two strategies can reduce the number of label updates by a factor of 10~30.

4.2 Summary of Label Computation

For a given an area bound, our labeling algorithm binary search the minimum clock period in a range from 1 to nD or a much smaller range computed by existing algorithms. For each target clock period, we compute all node labels and check if the label of each PO is no more than the target. It can be proved that the maximum number of labeling iterations is bounded by $B = n(n-1)D$. The total labeling time for each target clock period is $O(B \cdot K \cdot A \cdot log^2 D)$. The runtime to get the minimum clock period is $O(B \cdot K \cdot A \cdot log^2 D \cdot logn)$.

5 Duplication-free Cluster Formation

After getting the minimum clock period Φ_{min} and the corresponding node labels, one can construct the clustered circuit easily for nodes from POs to PIs with a first-in-first-out queue Q. Initially, we put all POs in Q. Each time we extract one node from Q and construct $C_{v,l^{opt}(v)}$ as presented in the previous section. Then, we put all inputs of $C_{v,l^{opt}(v)}$ in Q. This process is repeated until Q is empty. Finally we can perform a separate retiming on the clustered circuit based on the algorithms in [7, 10] to achieve a clock period of no more than $\Phi_{min} + D$. This procedure can be done in $O(A \cdot n + m)$ time, because we only form a cluster for each node once and the size of every cluster is bounded by A.

For k-way partitioning, however, the area overhead due to node duplication can be large even with some postprocessing of duplication removal. For example, the postprocessing proposed in [10] still results in 14% area overhead and 20% increase on the clock period based our experimental results with the program provided by the authors of [10]. Their postprocessing also takes very long time and is not applicable for large designs. As a result, we propose a performance-driven heuristic to construct duplication-free clusters. On average, we can achieve results with the same clock period as that by the approach in [10] and much smaller area in much shorter runtime. The details of our duplication-free

cluster formation is omitted due to page limit and can be found in [2].

6 Experimental Results

Cluserting with $A = \frac{n}{16}$					
		CLUS [10]		PRIME	
circuit	node	ϕ_L	t_L	ϕ_L	t_L
s208.1	77	11	0.2	11	0.0
s298	125	6	1.0	6	0.1
s344	122	13	5.6	13	0.1
s349	125	14	4.2	14	0.1
s382	150	7	1.3	7	0.1
s386	188	11	3.3	11	0.2
s420.1	165	13	1.0	13	0.0
s444	171	8	1.5	8	0.2
s510	213	14	8.8	14	0.7
s526	252	8	7.1	8	0.2
s820	468	14	29.5	14	0.4
s832	482	14	38.0	14	0.5
s838.1	341	16	5.7	16	0.1
s5378	1494	12	157.2	12	1.9
s9234.1	1313	20	305.7	20	5.6
s1196	481	21	4.6	21	0.0
s1423	508	52	59.0	52	0.8
s1488	734	15	56.0	15	1.0
s1494	746	15	63.8	15	1.3
s38417	9817	−	>20h	27	11.4
s38584.1	13292	−	>20h	29	8.1
bigkey	8607	−	core	7	2.2
clma	30556	−	ofm	54	1036.5
geo-sub	301	13	8.7	13	0.2
		+0%	37X	1	1
geo-mean	585	−	−	15	0.5

Table 1: Comparison of our labeling algorithm with CLUS [10] for 16-way partitioning. Item ">20h" means CLUS has been stopped after running 20 hours on a SUN Enterprise 4000 with 1.5GB memory without producing a result. Item "core" mean CLUS core dumpped for the example. Item "ofm" means CLUS run out of memeory on the SUN Enterprise 4000. "geo-sub" means the geometric mean of the first 19 examples. "geo-mean" means the geometric mean of all the examples.

We have implemented our algorithm, named PRIME, in the C programming language. The test set includes 23 IS-CAS benchmarks and 5 large industrial designs provided by an industrial sponsor. The experiments were run on a SUN Ultra2 workstation with 512MB memory. For designs with over 5000 gates the algorithm in [10] was run on a SUN Enterprise 4000 with 1.5GB memory.

To show the effectiveness of our algorithm for performance-driven k-way partitioning, we compared our algorithm with the algorithm in [10] with $k = 16$ and the area bound $A = \frac{n}{16}$. (We omit the results for clustering as our algorithm can be even faster.) The node delay is assumed to be 1 and the intercluster delay is 2 as in [10]. The results are shown in Table 1. Columns ϕ_L list the minimum clock period computed by labeling by the two algorithms, respectively. Columns t_L list the CPU time of the label computation. The results show that both algorithm can compute the same clock period, while PRIME algorithm is 37 times faster on average for designs with less than a few thousand gates. If we look at the last four large examples with over 5000 gates, the algorithm in [10] run either out of memory (over 1.5GB) or thousands times slower.

8 Acknowledgements

References

[1] J. Cong, H. Li, S. Lim, T. Shibuya, and D. Xu. Large Scale Circuit Partitioning With Loose/Stable Net Removal And Signal Flow Based Clustering. In *IEEE International Conference on CAD*, pages 441–446, 1997.

[2] J. Cong, H. Li, and C. Wu. *Simultaneous Circuit Partitioning/Clustering with Retiming for Performance Optimization*. UCLA-CSD 990019, Technique Report, March 1999.

[3] T. H. Cormen, C. H. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 25. The MIT Press, 1990.

[4] C. Fiduccia and R. Matheyses. A Linear-Time Heuristic for Improving Network Partitions. In *ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[5] L. Hagen and A. B. Kahng. New Spectral Methods for Ratio Cut Partitioning and Clustering. *IEEE Trans. on Computer-Aided Design of Integrated Circuits And Systems*, 11(9):1074–1085, 1992.

[6] E. L. Lawler, K. N. Levitt, and J. Turner. Module Clustering to Minimize Delay in Digital Networks. *IEEE Trans. on Computers*, 18:47–57, 1969.

[7] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[8] L. Liu, M. Kuo, C. K. Cheng, and T. C. Hu. Performance-Driven Partitioning using a Replication Graph Approach. In *Prod. 32th ACM/IEEE Design Automation Conference*, pages 206–210, 1995.

[9] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. On Clustering for Minimum Delay/Area. In *IEEE International Conference on CAD*, pages 6–9, 1991.

[10] P. Pan, A. K. Karandikar, and C. L. Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits And Systems*, 17(6):489–498, 1998.

[11] Y. C. Wei and C. K. Cheng. Towards Efficient Hierarchical Designs by Ratio Cut Partitioning. In *IEEE International Conference on CAD*, pages 298–301, 1989.

[12] H. Yang and D. F. Wong. Circuit Clustering for Delay Minimization under Area and Pin Constraints. In *ED&TC*, pages 65–70, 1995.

Comparison with Traditional 16-Way Partitioning					
		LR Algorithm [1]		PRIME-DF	
circuit	node	ϕ	cutsize	ϕ	cutsize
s208.1	77	24	65	14	58
s298	125	20	96	8	85
s344	122	24	77	17	77
s349	125	26	72	18	73
s382	150	17	75	11	83
s386	188	14	110	12	119
s420.1	165	21	66	17	89
s444	171	22	73	10	97
s510	213	23	135	17	160
s526	252	15	101	13	144
s820	468	22	141	17	189
s832	482	22	139	17	201
s838.1	341	30	73	20	125
s5378	1494	27	257	16	456
s9234.1	1313	38	230	22	460
s1196	481	34	215	27	226
s1423	508	85	95	55	188
s1488	734	24	241	20	316
s1494	746	22	243	19	334
s38417	9817	38	356	34	1545
s38584.1	13237	34	380	29	2431
bigkey	8607	11	105	10	1139
clma	30556	89	944	64	6087
big1	28009	56	1569	30	7509
big2	52301	519	3325	491	15068
big3	30227	330	1432	296	8243
big4	31407	284	581	165	5580
big5	101979	83	3712	73	24864
geo-mean	1258	36.9	220	26.4	471
		+39.6%	-53%	1	1

Table 2: Comparison with traditional methods for 16-way partitioning. Columns "ϕ" list the clock period on partitioning results. Columns "cutsize" list the k-1 cost between partitions. "geo-mean" means the geometric mean of all the examples.

We also compared our results with those by traditional partitioning algorithm (LR [1]) which were developed for minimizing the cut size only. This is mainly to show how much we can improve on the clock period and cost we may need to pay on the cut size. The LR algorithm is a recently reported state-of-the-art bi-partitioning algorithm which produces the *best* partitioning results among methods published in the literature on the commonly used MCNC benchmarks [1]. The comparison is shown in Table 2. The results by LR algorithm are listed in columns under "LR". Our results with duplication-free cluster formation are listed in columns under "PRIME-DF". Our test results show that the clock period by iterative LR algorithm is 39.6% longer with 53% less cutsize.[5] One reason for PRIME to have a larger number of cutting edges is due to the fact that PRIME-DF may generate results more than 16 partitions (17 or 18) in some cases, because it does not have direct control on the number of partitions to be generated as LR [1].

7 Discussions and Future Work

In this paper, we present an efficient and effective performance-driven clustering and partitioning algorithm for sequential circuits with retiming. Comparing with algorithm in [10], our algorithm is one order of magnitude faster if the area

---
[5] For a net spanning k partitions, its cutsize is $k - 1$ [1].