Power Minimization by Clock Root Gating

Qi Wang & Sumit Roy
Cadence Design Systems, Inc
River Oaks Parkway, San Jose, CA 95125, USA
qwang@cadence.com , sroy@cadence.com

Abstract -Clock root gating transformation targets power savings on the clock tree by inserting gating logic at the root of the clock. In this paper we propose an efficient graph-based algorithm to solve the root clock gating optimization problem. The algorithm is also tightly integrated with clock tree synthesis tool so that real power savings can be achieved after clock tree is generated. Experimental results on industrial circuits showed that significant power savings can be achieved.

I Introduction

The proliferation of portable consumer electronics, high-performance microprocessors has caused a major paradigm shift in the electronic design community. Power consumption has become one of the key concerns of designers.

Clock gating is a well-known technique to reduce dynamic power dissipation of a digital circuit [1,2,4]. It saves power by shutting off the sequential elements and part of the clock-network during the idle state. However clock gating saves power mainly on the registers and a small portion of the clock tree as the gating logic tends to be placed close to the registers [3]. Since a major portion of the clock tree is still toggling and drives large loads, the power consumed on the clock tree could be still very significant. Clock root gating transformation targets power savings on the clock tree itself. It inserts gating logic at the root of the clock to save the power consumed by the clock network feeding these instances. The clock tree is shut down when all the clock gating instances are disabled. Without loss of generality, for the rest of the paper we will refer clock root gating as simply *root gating*.

Fig. 1 illustrates a simple example with root gating potential. From Fig. 1, it can be seen that the clock gating logic instances (g1,g2,g3,g4) are placed close to the leaf register banks(r1,r2,r3,r4). As a result, the sub-clock-tree of b1 to b2 and b1 to b3 are still toggling and consumes power, although they do not have to toggle all the time. Assume the enable signal for g1 and g2 is a and the enable signal for g3 and g4 is $a'b$ and ab respectively. The enable signal for register bank r5 is a'. For sub-clock-tree b1 to b2, the root gating logic c1 is added. The enable signal is simply the enable signal a because for g1 and g2 the enable functions are the same. Because of root gating instance c1, the toggles on the sub-tree from b1 to b2 will be reduced. Similarly, for sub-clock-tree b1 to b3, the root gating logic c2 is added. The enable signal is the Boolean OR of the enable signals for g3 and g4, i.e. $a'b+ab=(a+a')b=b$.

As far as to our knowledge, there is no commercial tool offering automatic root gating transformation for power

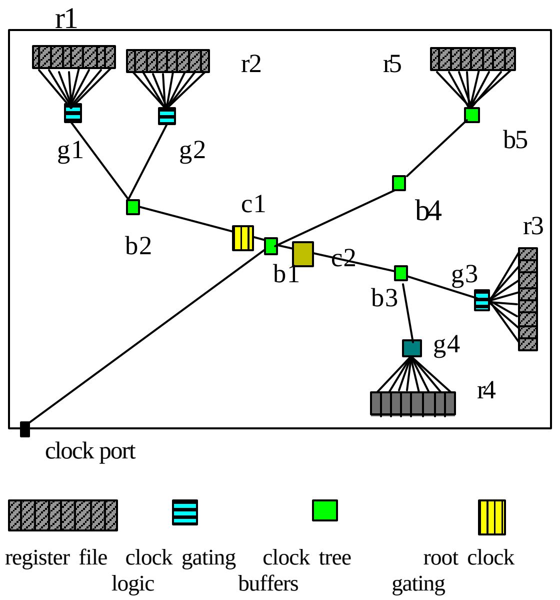

Fig. 1. Physical view of a simple clock tree.

optimization. Furthermore, most of the publications [2,3]on this topic suffer from the same shortcoming in that no physical information of the clock tree sinks is considered during the clock gating transformation. As a result, the clock tree constraints may not be met or the power of the design may actually go up after the clock tree synthesis [6].

II Problem of Root Gating

There are two major challenges in applying the root gating transformation in the optimization flow. First the possible number of candidates for root gating grows exponentially with the number of clock gating logic in the design. In fact if the total number of clock gating instances of a design is n then the total possible ways to root gate these n instances is $O(2^n)$. In a complicated digital design, there may be hundreds or even thousands of clock gating instances inserted during front-end synthesis. Therefore, an exhaustive search algorithm for the optimal solution of root gating is not feasible.

Second, applying root gating transformation without taking into account the physical placement information of the clock gating instances will impose a major challenge to clock tree synthesis tool. The fragmentation of the clock tree into several clock gating domain makes skew balancing very difficult [4,5]. Additionally, the inserted root gating logic may create unnecessary duplications of sub-clock tree. This may increase the clock tree insertion delay and power dissipation after clock tree has been generated.

In this paper, we present a novel graph based algorithm to achieve near optimal clock root gating insertion on a placed design. In Section III, we are introducing two basic data structures used by the proposed algorithm. Section IV describes the proposed algorithm in detail. An improved root gating scheme is presented in Section V. The experimental results on a few industrial designs are given in Sect ion VI. Finally the conclusion and future work is given in Section VII.

III. Basic Data Structures

A. *Clock-Gating Graph (CGgraph)*

Let *D* be the netlist representation of a digital design. A CGgraph $G_c(R, N)$ of a design *D* is a DAG (Directed Acyclic Graph) to represent the structure of the clock tree of the design. *N* is the set of nodes (called CGnode) and *R* is the root node of the graph. Each CGnode *n* has the following fields: *type, pin, enable, parent, and children*. For the rest of the section, we use n(.) to denote a field of CGnode n. Each CGnode has a corresponding instance pin in *D*, as defined by the field *pin*. If the *pin* is the clock pin of a clock gating instance, the field *enable* of the node is the enable signal for the clock gating instance. Otherwise it is null. Let *n*(pin) denotes the *pin* field of CGnode *n*. Then a CGnode *n* is the parent of another CGnode *m* if and only if there exists a path from *n*(pin) to *m*(pin) in *D*. There are five different types of CGnode as described below, where *n*(type) denotes the *type* of the CGnode *n*:

n(type) = **I** : The CGnode corresponds to the output pin *p* of a clock gating instance (e.g. AND gate) in *D* where *n(pin) = p*. *n(enable)* is the enable signal for the gating logic. For a latch based clock gating scheme, *n(enable)* is simply the data input of the corresponding gating latch. *n(parent) = m* where *m* is also a CGnode and *m*(type)=L or P or T. *n(children)* =? .

n(type) = **R** : The CGnode corresponds to the output pin *p* of a register in *D* where *n(pin) = p*. *n(enable)* is the enable signal for the register if there is any. *n(parent) = m* where *m* is also a CGnode and *m*(type)=L or P or T. *n(children)* =? .

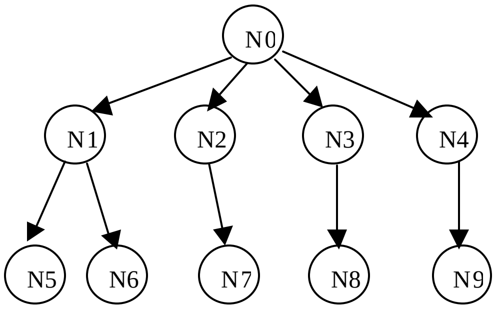

Fig. 2.1. The CGgraph for the design in Fig. 1 with logic clustering only.

n(type) = **L** : This CGnode is a Logic Clustering Node (LCN) which has no corresponding object in *D* and *n*(pin) = *null*. It represents a partition of the set of CGnodes C_{IR} of type I and/or R, which have the same enable function. *n(enable)* is the common enable function of its children where *n(children)*= G_R and *n*(parent) = *m*. *m* is also a CGnode and *m*(type)= P or T.

node name = {type, pin, enable, parent, children}

N0 = {T, clk, null, null, {N1,N2,N3,N4}}
N1 = {L, null, a, N0, {N5,N6}}
N2 = {L, null, a’, N0, {N7}}
N3 = {L, null, a’b, N0, {N8}}
N4 = {L, null, ab, N0, {N9}}
N5 = {I, g1, a, N1, null}
N6 = {I, g2, a, N1, null}
N7 = {R, r5, a’, N2, null}
N8 = {I, g3, a’b, N3, null}
N9 = {I, g4, ab, N4, null}

Fig. 2.2 The detail information of each node in Fig. 2.1

n(type) = **P** : This CGnode is a Physical Clustering Node (PCN) which has no corresponding object in *D* and *n*(pin) = *null*. It represents a partition of the set of CGnodes C_L of type L whose children (I or R type nodes) fall into the same physical partition determined by their physical proximity. *n*(children) = C_L. *n*(parent) = *m* where *m* is also a CGnode and *m*(type)= P or T. *n*(enable) = *null*.

n(type) = **T** : The CGnode corresponds to the root pin *p* of a clock tree in *D* where *n*(pin) = *p*. *n*(enable) = *null*. *n*(parent) = *null* and *n*(children) is a set of nodes of types L, P, I, or R.

The CGgraph $G_c(R, N)$ for the example in Fig. 1 without considering physical proximity of the gating instance is shown in Fig. 2.1; where *N*={N0,…,N9} and *R*=N0. The detail information for each node is shown in Fig. 2.2. For example, for node N5 the corresponding object in the netlist is the out pin of g1; the type is I (gating instance) and the enable function of the gating logic is *a*; the parent of the node is N1 and it has no children. Similarly, node N1 is a logic cluster node with enable function of *a* because both of its children N5 and N6 having the same enable function. It can be seen that the logic cluster node represents the leaf clock gating instances of a design with the same enable function. Now the task of root gating is to identify the best set of LCN’s such that the combined root gating logic will reduce the power of the clock-tree.

B. *Root-Gating Graph (RGgraph)*

An RGgraph $G_r(N,E)$ is an undirected graph used to explore possible combination of different LCN’s of a CGgraph to identify candidates of root gating. It consists of sets of nodes *N* (RGnode) and sets of edges *E* (RGedge).

An RGnode n is a tuple of (M,f_e) where *M* is a set of

CGnodes $m_1, m_2, \ldots, m_k$ where f_e is the combined Boolean OR of the enable functions of nodes in M and $f_e = m_1(enable) + m_2(enable) + \ldots + m_k(enable)$.

An RGedge e is an edge between two RGnodes n_{left} and n_{right} where the enable function f_e of the edge is simply $f_e(n_{left}) + f_e(n_{right})$.

Simply put, each RGnode represents a set of CGnodes that can be grouped together to form a candidate of root gating. The RGedge represents another root gating candidate by gating the RGnodes it is connected to.

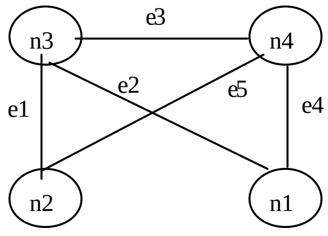

Fig. 3. The RGgraph built from the CGgraph shown in Fig. 2.

Let us explain the construction of an RGgraph, as shown in Fig. 3, for the CGgraph in Fig. 2.1. In Fig. 2.1, there are 4 CGnodes with type of L. Therefore in the RGgraph, 4 RGnodes {n1,n2,n3,n4} are created in Fig. 3 corresponding to the CGnodes {N1,N2,N3,N4} in Fig 2.1. For example, for node n1 in Fig. 3, the corresponding CGnode is N1 in Fig. 2.1. The enable function of n1 is the enable function of N1 which is a. An RGedge represents a possible grouping of the left and right RGnode to form a new candidate for root gating. For example the edge e3 represents a possible grouping of N3 and N4 to form a root gating. The enable function of e3 is the Boolean OR of the enable functions of the nodes connected by the edge, i.e. $a'b+ab=b$. Note that, if the enable function is 1 then no possible power saving can be achieved by using the enable. As a result, there exists an edge between two RGnodes if and only if the OR function of the enable function of the two nodes is not 1. For example, there is no edge between n1 and n2 because the OR function of the enable function of n1 and n2, i.e. a and a' respectively, is $a+a'= 1$.

Since an RGedge represents a candidate for a root gating move, the RGgraph represents all possible pair wise root gating candidates of all RGnodes. New RGnodes can be created by merging the RGedges. The merging of an RGedge means the acceptance of root gating of the nodes it connected to save power. The enable function of the new node is the enable function of the merged edge. The estimated amount of the power saved by merging an edge is the product of the probability of the enable function of the new node being off and the power consumed by the subtree. With the new RGedge created, the edge set of the RGgraph needs to be updated based on the original connectivity of the merged edge. For example, the edge merge process of the RGgraph in Fig. 3 is shown in Fig. 4. For exemplary purpose, we assume the power saving can always achieved as long as the signal probability of the enable function is not 1. First node n3 and n4 are merged. The new merged node is n34 and the new enable function is $ab+a'b=b$. All the edges connected to n3 and n4 are removed. However, since n2 connected to both n3 and n4 before, a new edge e6 will be created between the new node n2 and n34 to represent the potential root gating of merging these two nodes in the future. The enable function is the OR of the enable of n2 and n34, i.e. $a'+b$. Similarly, edge e7 is created with enable function $a+b$. The next merge is for edge e6 that merges n34 and n2. A new node n342 created with enable function $a'+a'+b=a'+b$. There will be no edge between n342 and n1 although n1 is connected to n34 before. This is because the enable function of the edge is $a'+b+a=1$. The merge process stops since no more edges exist in the graph.

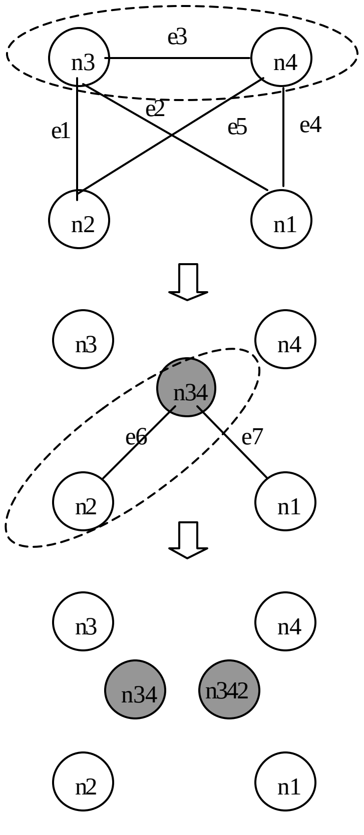

Fig. 4. Merge of RGgraph in Fig. 3.

With the RGgraph, the task of finding the best sets of LCN's such that the com bined root gating logic will save the power of the clock tree becomes the exploration of the RGgraph by iteratively merging the edges until no more edge in the graph. The final RGgraph is a collection of RGnodes and each node represent a candidate of root gating.

An important property of the RGgraph is that there is an edge between two nodes if and only if the OR'ed enable function of the nodes it connected to is not 1. In practice, the constraint can be further tightened such that there is an edge if and only of the signal probability of the OR'ed enable function of the nodes it connected to is not significantly close to 1. For example, in Fig. 4, after the first merge, the enable function of edge e6 becomes a'+b. If the signal probability of the function is very close to 1, we know that merging node n34 and n2 is not a good candidate. We can simply remove the edge and the merge process stops right there. Therefore even though the RGgraph may consist of a lot of edges initially, the merge process is very fast because the number of mergeable edges drops very quickly.

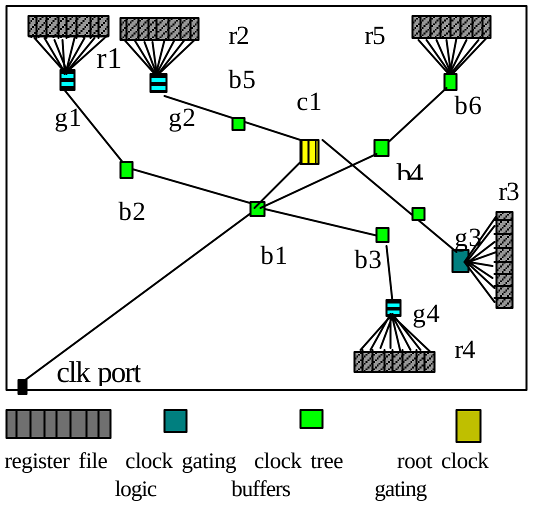

Fig. 5. An example of bad root gating - without considering the physical placement

C. *Physical Partition of clock gating instances*

As pointed out earlier, one of the major challenges in applying root gating transformation is that it has to be closely integrated with the clock tree synthesis tool and take the physical placement into consideration. For the example shown in Figure 1, assume the probability of the combined enable function for g3 and g4 is very low. The lower the probability of an enable signal, the higher chance for power savings when it is used to gate clock. Therefore without taking the placement information into account, we may consider root gating g2 and g3, as shown in Figure 5. It can be seen clearly that this root gating will create a lot of duplication in the clock tree. Therefore instead of saving power it may actually increase power, compared to the clock tree in Figure 1. In this work, we will use a fast mode clock tree synthesis to produce a reasonable realistic clock tree. We will then use this tree to create a physical partition of the clock gating instances that are the targets for root gating. The advantage of this approach is that this physical partition is very close to the final synthesized clock tree. When the physical partition information becomes available, the power saving estimated during root gating exploration will be very close the result seen after clock tree synthesis. Thus it prevents the situation illustrated in Figure 5 from happening. For example, assume Figure 1 represents the netlist after fast clock tree synthesis, the corresponding CGgraph with physical clusters nodes is shown in Figure 6.

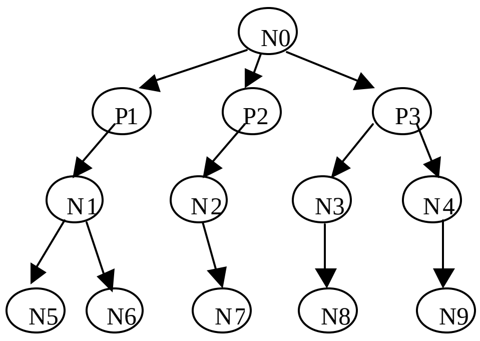

Fig. 6. The CGgraph of Fig. 2.1 when physical cluster nodes are inserted given the netlist in Figure 1.

```
rootGating (D) {
/* D  is a gate level netlist */
1. oldPow = estimated clock tree power for D;
2. C = set of all leaf clock gating instances;
3. R = set of all regs with enable function;
4. $G_c$ = constructCGgraph(C,R);
5. construct LCN and PCN for $G_c$;
6. $G_r$ = constructRGgraph($G_c$);
7. mergeRGgraph($G_r$);
8. committRGgraph($G_r$);
9. buildClockTree(D);
}
```

Fig. 7. Root clock gating algorithms.

In Figure 6, the nodes N5 to N9 are partitioned into three physical clusters. For example, N5 and N6 belong to physical cluster P1. N7 belongs to physical cluster P2. N8 and N9 belong to the physical cluster P3. Only CGnodes within the same physical cluster will be considered for root gating. In other words, a new RGgraph is created for each physical clustering node and root gating exploration only happens within that physical partition.

IV. The Algorithm

A. *Overall Algorithm*

The overall algorithm for the root gating optimization is shown in Fig. 7. A key concept of the algorithm is that the separation of the trial of each root gating move and the actual commit (implementation) of the move. The advantage of this approach is that during trial phase a simpler but much faster cost function can be used to explore all possible moves. Then during commit phase, a more accurate cost function can be used to determine the power savings and slack overhead for each move. If a move saves power and does not violate timing constraints, the move will be accepted and committed. The trial of potential root gating move in the algorithm is simply the exploration RGgraph by merging the edges until no edge left (mergeRGgraph). The result of the exploration is a set of RGnodes where each node corresponds to a potential root gating move. Then in the decreasing order of estimated power savings for each possible move (i.e. RGnode), each move will then be committed in the later stage.

The details of step 4 through 6 have been covered in earlier sections. At step 5, a fast mode of clock tree synthesis is called and the resulting CGgraph contains both LCN and PCN for later use. Due to the limitation of the size of the paper, in the following sections, we will only discuss step 7 and 8 in greater detail.

B. Merge of RGgraph

The process of merging edges of an RGgraph is shown by an example in Section III.C. A formal description of the merging algorithm is shown in Figure 8.

The function estimatePowerSaving is to estimate the clock tree power to be saved by using the enable signal from the RGedge E. As mentioned in previous section, at this trial phase, a simple cost function can be us ed to evaluate the power savings. For example, it can be the product of the probability of the enable function of edge E being zero and the total power consumes by the clock gating instances. As a result, the lower the probability of the enable signal, the larger potential for power savings.

There are several key points to be noted in the above algorithm. For each edge of *Gr*, we try the root gating for all the corresponding nodes connected to the edge. If power saving is greater than 0 it means the current root gating may be a good candidate. Then a new node is created by merging the edge to represent the solution. The power saving of the new node is the power saving evaluated from the edge. The enable function of the new node is the enable function of the merged edge. Note that a new edge is created only if the new enable function is not a constant 1 because if the new enable function is 1 then there is no chance to reduce the switching activity of clock using this enable function. Finally, at step 2 the order of the edges being selected is important. In our implementation, the edges are selected in the decreasing order of its estimatePowerSaving.

```
mergeRGgraph ( Gr , D) {
/* D is the target design and Gr is the RGgraph */
1. while (number of edges of Gr > 0) {
2.    pick an edge E from Gr and delete it from Gr;
3.    powerSaving = estimatePowerSaving(E,D);
4.    if(powerSaving > 0) {
5.       $N_{new}$=create new RGnode;
6.       $N_{new}$[enable] = E[enable].
7.       LR = all nodes conn. to the left/right node of E;
8.       delete all edges conn. to the left/right node of E;
9.       for each (node $N_{LR}$ in LR) {
10.          en = $N_{LR}$[enable] OR $N_{new}$[enable];
11.          if(en != 1) {
12.               $E_{new}$=createNewRGedge($N_{LR}$,$N_{new}$);
13.               $E_{new}$[enable] = en;
14.          }
15.      }   /*end of for loop*/
16.      $N_{new}$ [powerSaving] = powerSaving;
17.   }   /*end of while loop*/
18. }
```

Figure 8. Algorithm of mergeRGgraph

C. Commit RGgraph

The result of a merged RGgraph is an RGgraph with a set of nodes but without edges. The exhaust of edge means no more new root gating candidate can be found to possibly reduce power dissipation. Also each RGnode, which corresponds a set of gates in the netlist, represents a good candidate for insert root gating in front of those gate objects. Each RGnode has a cost field that represents the estimated power saving for each root gating transformation. The cost is updated during the merge process. The root gating decommit algorithm is shown in Fig. 9.

At the beginning of the commit phase, all nodes of the merged RGgraph are sorted in the decreasing order of power savings. Since each of the RGnode represents a potential root gating candidate, the top of the list is the candidate with largest power savings. Then a RGnode is taken from the top one by one and a root gating logic is inserted using the enable function associated with the RGnode.

Inserting root gating logic involves two basic operations. First the composite enable logic needs to be created. It is the Boolean OR of the enable functions of all the leaf gating logic to be tried for this root gating move. Because the enable function for each leaf gating logic may be presented in different logical hierarchy, new ports may be created to for the new enable signal. Secondly, the new clock signal is connected to the clock pins of the leaf gating logic. For the same reason, new ports may also be created.

VI. Partially Committing of Root Gating

As described above, the enable signal to root gate a set of objects is the Boolean OR function of the enable signals of these objects. However, the additional logic will add delays of the clock signal, which may cause timing constraint violation. Instead of not committing the root gating transformation when it occurs, another choice is to see if the enable function can be simplified to reduce the delay. For example, let the enable function for the root gating logic be $ab+bc=a(b+c)$. Assume that signal b is very late, the added logic of $a(b+c)$ may significant delay the clock signal and hence may worse the performance of the root gated design. However we know that signal a itself can be used to root gate

```
commitRGgraph (D, Gr) {
/* D is the target design and Gr is the RGgraph */
1.  S = sub set of all Gr nodes whose power saving
      value is greater than 0;
2.  sort all nodes in S in the decreasing order of
     power saving;
3.  while (S is not empty) {
4.      pick the top Nr from S;
5.      C = all the gate objects referred by Nr;
6.     insertRootGating(C,D);
7.   } /*end of while loop*/
8. }
```

Figure 9. Algorithm of commitRGgraph

those clocks. Although by doing this, the probability of the clocks being shut off is reduced but the logic of the enable signal is simplified. Since signal b is no longer part of the root gating enable function, the design will be faster than the one using *a(b+c)* as the enable function. To achieve this improvement, only a minor modification is required in the algorithm shown in Fig. 7. In step 8 of Fig. 7, after committing a root gating transformation, the design is checked for possible timing violation introduced by the root gating logic. If so a new enable function is searched for the root gating logic and used if found one.

VI. Experimental Results

The proposed algorithm is implemented in Cadence$^{?}$ PKS/LPS 5.0 release [7]. The PKS/LPS flow starts from the RTL of a design. During generic optimization, clock gating logic is inserted. It is then followed by pre-placement timing and power optimization. Root gating transformation is applied after placement but before clock tree synthesis. Finally, PKS optimize the timing and power simultaneously.

Four industrial circuits with real timing and clock constraints and physical floorplan are selected for experiment. Circuits A and B are control logic. Circuit C is a DSP block and Circuit D is a communication chip. For C and D, customer test benches are used to run simulation to generate the switching activity file for power estimation [7]. For design A and B, random vectors are used. All experiments are run on a SUN workstation with 750 MHz CPU and 4 GB of memory. The experimental results are shown in Table 1.

The column "Total Inst." shows the total number of instances in each design. The columns "Flip Flops" and "Clock Gated" show the total number of sequential elements and the number of clock gating inserted by LPS [7], respectively. The column "% CT Power" is the power of clock tree as a percentage of the total power. The next three columns show the clock tree power, insertion delay and skew respectively. The sub-columns "w/o RG" show the results for running the test cases with normal PKS low power flow [7]. The sub-columns "w/ RG" show the results for running the test cases with the root gating transformation. The columns "%" show the percentage improvements for the results. The last column shows the number of root gating inserted for each circuit.

From the table it can be seen that root gating can save significant clock tree power on big designs. The impact of the transformation on the clock skew is very marginal however the maximum clock insertion delay becomes worse for all cases. For small designs, the results of applying root gating are mixed. For example, for design A, the clock tree power actually increases a little bit after the optimization has been done. A detailed analysis showed that the fast clock tree estimation during root gating transformation does not correlate well with the final synthesized clock tree for relatively small clock trees. The run time and memory usage overhead of the additional root gating transformation in the design flow is very little. Also there is no significant difference in the circuit worst slack for the flow with and without root clock gating. Due to the limitation on the size of the paper, these results are not shown here.

VII. Conclusions and Future Work

In this paper, we have proposed a novel solution for root gating optimization that further reduces the clock tree power. The novelty lies in using efficient graph-based data structures and algorithm to solve the problem and making the optimization knowledgeable of placement of clock tree and the topology of clock-tree.

Future work in this area are: (1) improve the physical partition algorithm to achieve a fast yet accurate physical partition of all clock gating logic; (2) improve the integration with clock tree synthesis to reduce the clock insertion delay after root gating inserted; (3) implement post clock tree synthesis root gating de-commitment. The idea is that, after clock tree synthesis, if it is found that root gating violates timing or clock tree constraints, the tool should be able to remove them. In this case, incremental clock tree synthesis rather than a full clock tree rebuild is needed.

References

[1] L. Benini and G. De Micheli, "Automatic synthesis of low power gated clock finite state machine," IEEE Trans. Computer-Aided Design, vol. 15, pp. 630-643, June 1996

[2] A. Farrahi, C. Chen, A. Srivastava, G. Tellez, and M. Sarrafzadeh, "Activity driven clock design," IEEE Trans. Computer-Aided Design, vol. 20, pp. 705-714, June 2001.

[3] J. Oh and M. Pedram, "Gated clock routing for low power microprocessor design,", IEEE Trans. Computer-Aided Design, vol. 20, pp. 715-722, June 2001.

[4] D. Garrett, M. Stan and A. Dean, "Challenges in Clock-gating for a Low Power ASIC Methodology," International Symposium on Low Power Electronic Design (ISLPED), pp. 176 – 181, August 1999.

[5] T. Kitahara and et. al., "A Clock-Gating Method for Low Power LSI Design," International Symposium on Low Power Electronic Design (ISLPED), August 1998.

[6] D. Garrett, "A Low Power Normalized-LMS Decision Feedback Equalizer for a Wireless Packet Modem", Proceeding of ISLPED, August 2002.

[7] Cadence$^{?}$ Low Power Synthesis (LPS) User's Guide for Cadence$^{?}$ PKS.

	Total	Flip	Clock	% CT	**CT Power (mW)**			**CT Delay (ns)**		**CT Skew (ns)**		RG
Design	Inst.	Flops	Gated	Power	w/o RG	w. RG	%	w/o RG	w. RG	w/o RG	w. RG	Inserted
A	6519	1282	40	66.8%	1.15	1.19	-3.5%	3.12	3.07	0.48	0.51	1
B	9523	2340	40	55.8%	5.23	5.11	2.3%	4.70	5.62	1.07	0.86	4
C	23539	1960	38	89.7%	46.27	40.62	12.2%	1.10	1.20	0.11	0.13	3
D	63121	5087	303	54.3%	79.00	72.57	8.1%	1.27	1.75	0.30	0.30	17
Avg.	25676	2667	105	0.67	32.91	29.87	**4.8%**	2.55	2.91	0.49	0.45	6

Table 1. Experimental results.