Post-Placement Voltage Island Generation *

Royce L.S. Ching, Evangeline F.Y. Young, Kevin C.K. Leung
Department of CSE
The Chinese University of Hong Kong
{lsching,fyyoung,ckleung}@cse.cuhk.edu.hk

Chris Chu
Department of ECE
Iowa State University
cnchu@iastate.edu

ABSTRACT

High power consumption will shorten battery life for handheld devices and cause thermal and reliability problems. One way to lower the dynamic power consumption is to reduce the supply voltage. Multi-supply voltage (MSV) is introduced to provide higher flexibility in controlling the power and performance trade-off. In region-based MSV, circuits are partitioned into "voltage islands" where each island occupies a contiguous physical space and operates at one supply voltage. In a very recent work [6], this supply voltage partitioning problem is addressed, and the input circuit is partitioned into a slicing structure with every voltage island rectangular in shape. This unnecessary restriction on the structure and island shapes has caused a significant degradation in the solution quality. In this paper, we propose a method to solve this voltage island generation problem without these restrictions. Experimental results have shown that our approach is fast and can improve the solution quality significantly. In some data sets, only two voltage islands are needed to satisfy the same power consumption bound while the approach in [6] will generate nineteen.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; J.6 [**Computer Applications**]: Computer-aided design—*Computer-aided design (CAD)*

General Terms

Algorithm, Design

Keywords

Voltage Island, Floorplanning, Tree

1. INTRODUCTION

The high functionality of SoC designs today and the much higher leakage current are leading designs to power dissipation of hundreds of watts. High power consumption will shorten battery life for handheld devices and cause thermal and reliability problems. This power dissipation problem is expected to get worse at the future process nodes. There are two sources of power consumption: dynamic and static [2], and the former has dominated the total power consumption in today's designs. Dynamic power consumption can be reduced by techniques like reducing switching activity, load capacitance and supply voltage. Reducing supply voltage can significantly lower the dynamic power consumption which is proportional to the square of the supply voltage. Multi-supply voltage (MSV) is thus introduced to provide higher flexibility in controlling the power and performance trade-off. There are two types of MSV, row-based and region based. In row-based MSV, standard cells of high and low supply voltage will interleave in different rows. In region-based MSV, circuits are partitioned into "voltage islands" where each island occupies a contiguous physical space and operates at one supply voltage [1, 4, 3].

Region-based design is mostly done manually based on a design's logic hierarchy. However, logic boundaries may not be good for voltage supply partitioning. These logic boundaries are usually non-optimal for supply voltage partitioning [6]. Besides the power issue, it is also important to minimize the number of voltage islands to reduce the overhead in voltage shifting devices and the cost in routing the power networks.

Wu *et al.* [6] is the first to consider this supply voltage partitioning problem capturing the power versus design cost (number of voltage islands) trade-off. They partition the input circuit into a slicing structure in which every voltage island is rectangular in shape. This unnecessary restriction on the structure and island shapes has caused a significant degradation in solution quality. In some data sets tested in [6], only two voltage islands are needed to satisfy the same power consumption bound while the approach in [6] will give 19 voltage islands. In this paper, we propose a method to solve this voltage island generation problem without this restriction on island shapes. Experimental results have shown that our approach is very efficient and can give much better results than [6] in terms of reducing power consumption and the number of islands. The improvement in solution quality is mainly due to the relaxed restriction on the island shapes. In order to demonstrate the effectiveness of our approach, we have also compared our method with a greedy heuristic which tries to solve the problem directly and we can see that our approach is also better in terms of both solution quality and run time.

In the following, we will define the problem in section 2, then we will discuss the methodology used in our approach in section 3. In section 4, we will describe the greedy heuristic, and the experimental results will be reported in section 5.

*The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4188/03E.)

ICCAD'06, November 5-9, 2006, San Jose, CA

2. PROBLEM FORMULATION

In this voltage island generation problem, we are given an $m \times n$ grid based placement P of N cells where each cell has its own voltage requirement, and the objective is to partition this placement into a set of contiguous regions such that the total number of regions is as small as possible, and when each region is supplied with the same voltage, i.e., the maximum voltage in that region (since the voltage assigned to a cell should not be lower than its required voltage), the total wastage in power does not exceed a given threshold. We will use an $m \times n$ array A to represent the dynamic power required at each point. Notice that dynamic power is proportional to the square of the supply voltage. Each cell will occupy an integral number of elements in A (we use the word "element" to describe a room inside a grid) and there may be empty elements in between. An example is shown in fig. 1. The voltage island generation problem is to partition this array into a set of connected regions $\Pi = \{R_1 \dots R_k\}$ [1] such that the size of the partitioning $|\Pi| = k$ is as small as possible and the total power wastage $w(\Pi)$ does not exceed a given threshold where $w(\Pi)$ is defined as:

$$w(\Pi) = \sum_{1 \le t \le k} w(R_t),$$

the power wastage $w(R)$ of a region R is defined as:

$$w(R) = \sum_{(i,j) \in R} (\mu(R) - A[i,j])$$

and $\mu(R) = \max_{(i,j) \in R} A[i,j]$ is the maximum power required in region R.

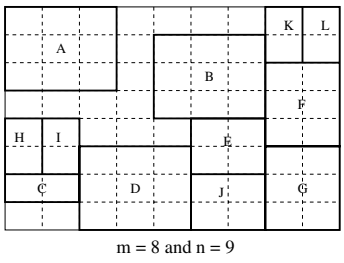


Figure 1: An Example of the Input Grid Based Placement.

This problem is similar to the Voltage-partitioning Problem (VPP) defined in [6], except that the regions are not restricted to be rectangular in shape. We formally define our problem as follows:

Non-rectangular Voltage-partitioning Problem (NVPP) Given an $m \times n$ array A and an error threshold δ, find a partitioning Π of connected regions whose weight $w(\Pi)$ is at most δ and the size $|\Pi|$ is as small as possible.

3. METHODOLOGY

The input grid size $m \times n$ is usually huge and it is very inefficient to work on it directly. Therefore, we will first coarsen the grid to a size of $O(N)$ where N is the number of cells. We will then build a tree T of the elements in the coarsened grid according to their adjacencies and their differences in power requirement. A dynamic programming will be applied to partition this tree optimally into subtrees (each subtree will correspond to a connected region) such that the total power wastage is minimized. This tree T will be reconstructed and partitioned repeatedly until the total power wastage reaches the given threshold. An overview of the whole process is shown in fig. 2 and details of each step will be given in the following sections.

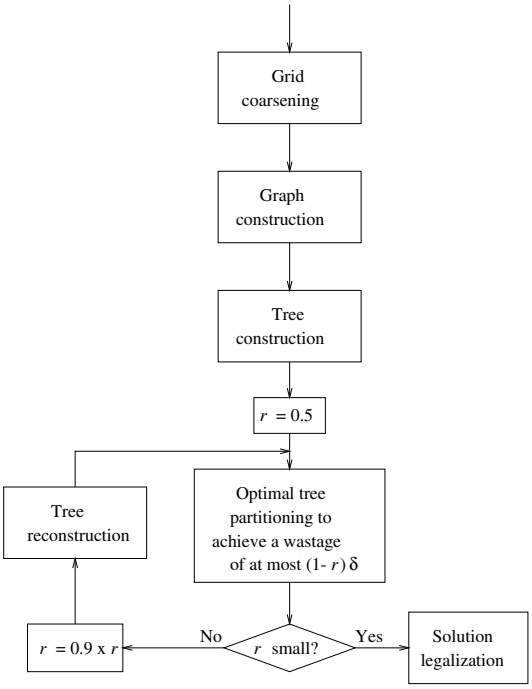


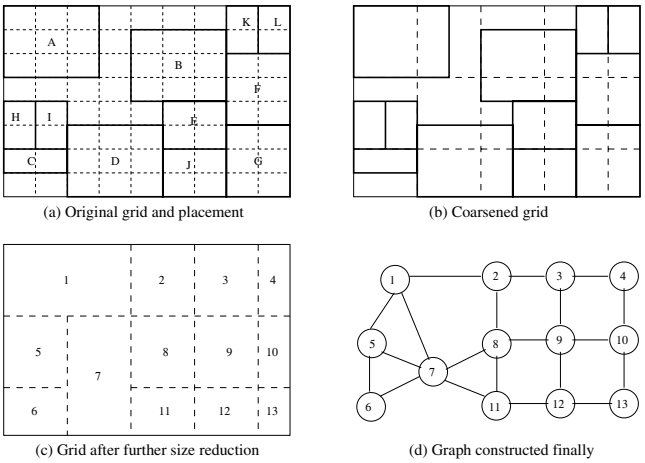
Figure 2: An Overview of Our Approach.

3.1 Grid Coarsening and Graph Construction

Given an $m \times n$ grid-based placement P, we will first coarsen the grid into one with about N square-like shaped elements where N is the total number of cells in P. This can be done by dividing the original grid into rows and columns of elements with heights and widths approximately equal to $\sqrt{\frac{W \times H}{N}}$ where W and H are the width and height of P respectively. Notice that we will adjust the widths and heights of the rows and columns in the coarsened grid slightly such that every element in the original grid will be in one and only one element of the new grid. An example is shown in fig. 3(b). Let this coarsened grid be A' with size $m' \times n'$. Now, each element $A'[i,j]$ in the coarsened grid A' will contain a set of elements in the original grid A and the value of $A'[i,j]$ will be set to the maximum value among these elements in A. Notice that this coarsening step has already caused a small amount of power wastage and the function $cost(x)$ is used to denote the power wastage brought at an element x in A' due to this coarsening step. To further reduce the size of this coarsened grid, we will combine those neighboring elements that belong "completely" to the "same cell" to form one element. Notice that a cell cannot be split between islands at the end, so this enhancement is a reasonable step to reduce the size of A'.

After this coarsening step, we will construct a graph $G(V,E)$ from A' such that each element in A' will become a node in V and two nodes u and v in V will have an edge $e(u,v)$ between them if they are neighbors of each other in A'. An example of this graph construction step is shown in fig. 3(d).

[1] In terms of connectivity, it is assumed that each element is connected to its four neighbors.



(a) Original grid and placement

(b) Coarsened grid

(c) Grid after further size reduction

(d) Graph constructed finally

Figure 3: Grid Coarsening and Graph Construction.

Each node u in G will have a cost $cost(u)$, a size $size(u)$ that denotes the number of A's elements contained in u and a value $\mu(u)$ that denotes the maximum power required at u. Now the problem becomes a graph partitioning problem in which we want to partition G into a set of connected subgraphs $\{G_1 \dots G_k\}$ such that k is as small as possible and the total cost $\sum_{1 \le i \le k} cost(G_i)$ does not exceed a given threshold where the cost of a subgraph G_i is defined as:

$$cost(G_i) = \sum_{u \in G_i} ((\mu(G_i) - \mu(u)) \times size(u) + cost(u))$$

and $\mu(G_i) = \max_{u \in G_i} \mu(u)$ is the maximum power required in G_i.

3.2 Tree Construction

After constructing the graph G, we will perform a bottom-up clustering of G recursively. At each level, pairs of adjacent nodes are clustered to form a "super-node" at the next level according to the amount of power wastage incurred. A binary tree T is constructed to represent this multi-level clustering process, in which the leave nodes are the nodes in G and the internal nodes are the super-nodes. The pseudocode for this tree construction step is given as below:

Pseudocode *BuildTree()*
// Given a graph $G(V, E)$ where each node u in V has attributes
// $cost(u)$, $\mu(u)$ and $size(u)$, perform a bottom-up multi-level
// clustering on G and construct a binary tree T to represent
// this recursive clustering.

1. $S = V$.
2. *Repeat*
3. $S' = \emptyset$.
4. *Repeat*
5. *Find the nodes in S that has the smallest number of neighbors in S. Call this subset of nodes C.*
6. *Find a pair of nodes u and v such that $u \in C$, v is a neighbor of u and the clustering cost $a = (\mu(x) - \mu(y)) \times size(y)$ is the minimum where $x = u$ ($x = v$) and $y = v$ ($y = u$) when $\mu(u) \ge \mu(v)$ ($\mu(u) < \mu(v)$).*
7. *Create a node w in S' that represents a cluster between u and v.*
8. $\mu(w) = \max\{\mu(u), \mu(v)\}$, $size(w) = size(u) + size(v)$ *and* $cost(w) = cost(u) + cost(v) + a$
9. *Remove u and v from S.*
10. *Add two tree edges from w to u and v.*
11. *Until no more clusterings can be done.*
12. *Put all the remaining nodes in S to S'.*
13. $S = S'$.
14. *Until only one node left in S.*
15. *Put the only node in S as the root of T.*

In step 7 of *BuildTree()*, we will try to cluster those nodes with the smallest number of neighbors first in order to reduce the number of "dangling" nodes that cannot be paired up with any other nodes.

3.3 Optimal Tree Partitioning

After constructing T, we will partition it into a set of subtrees $\{T_1 \dots T_k\}$ such that every leaf node is contained in one subtree, k is the smallest possible, and the total cost $\sum_{1 \le i \le k} cost(r_i)$ where r_i is the root of T_i is smaller than a given threshold. This tree partitioning problem can be solved optimally by dynamic programming and the pseudocode is given below. Notice that after this tree partitioning step, each sub-tree corresponds to a connected region in the original placement P and the total cost $\sum_{1 \le i \le k} cost(r_i)$ is equivalent to the total power wastage of these corresponding regions in P.

Pseudocode *PartitionTree()*
// Given a threshold δ' and a binary tree T where each node u in
// T has an attribute $cost(u)$, partition T into a set of subtrees
// $\{T_1 \dots T_k\}$ (with roots $r_1 \dots r_k$ respectively) such that
// every leaf node is contained in one subtree, the total cost
// $\sum_{1 \le i \le k} cost(r_i)$ is within the threshold δ' and k is the
// smallest possible.

1. $k = 0$
2. *Repeat*
3. $k = k + 1$.
3. $cost = DP(T, k)$.
4. *Until* $cost \le \delta'$.
5. *Return the roots of the partitioned subtrees.*

Pseudocode *DP(T, k)*
// Partition a binary tree T, where each node u in T has an
// attribute $cost(u)$, into k subtress such that every leaf node is
// contained in one subtree and the total cost $\sum_{1 \le i \le k} cost(r_i)$
// is the smallest possible where r_i for $1 \le i \le k$ are the roots of
// the subtrees.

1. *Let r be the root of T.*
2. *If k is 1, return($cost(r)$).*
3. *If r has only one child, $DP(child(r), k)$,*
4. *Else*
5. $min = \infty$ *and* $k' = 0$.
6. *For* $i = 1$ *to* $k - 1$
7. $c = DP(lchild(r), i) + DP(rchild(r), k - i)$.
8. *If* $min > c$, $min = c$ *and* $k' = k$.
9. *Record k' in a table for later retrieval of the subtrees.*

3.4 Tree Reconstruction

We do not stop after one tree partitioning because the structure of the tree T (which represents the neighborhood of the leaf nodes) is very much dependent on the criteria we have used in *BuildTree()* to cluster the nodes. We will iteratively reconstruct the tree and re-partition it. The tree reconstruction step is exactly the same as the *BuildTree()* procedure except that we start with $S = \{r_1 \dots r_k\}$ where r_i is the root of a subtree obtained after partitioning in the last iteration. This tree reconstruction step is very important since it allows us to redistribute the nodes between the subtrees. Notice that after a tree T is reconstructed to T', the optimal partitioning of T will also be one possible partitioning of T'. Therefore the optimal partitioning solution of T' will not be worse than that of T. We will also progressively loosen the required threshold in total cost by

reducing the r in fig. 2, because it is better to produce more subtrees at the beginning (by having a smaller threshold) and gradually refine the solution by combining the subtrees (by partitioning with a larger threshold). In our implementation, r is set to 0.5 initially (note that $0 \leq r \leq 1$). A larger initial value can be used without degrading the solution quality and the value of 0.5 is found from experiments to be a good balancing point between run time and solution quality.

3.5 Solution Legalization

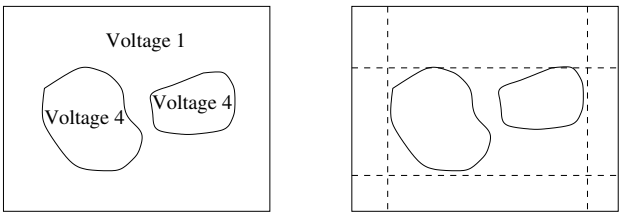
In the final solution of the above iterative reconstruction and partitioning step, there may be cells partitioned in more than one regions, which is not legal. Therefore we will perform a post-processing step to assign each of those cells to a region that have taken the largest portion of it. This legalization step is not performed in [6] but the difference incurred in the total power wastage is very small.

3.6 Time Complexity

In the coarsening step, we need to do one update for each intersection between a cell and an element of the coarsened grid A'. Each element in A' will intersect with at most a constant number of cells, so the time complexity is $O(m'n')$. Since we have put $m' \times n' \approx N$, the complexity of this step is $O(N)$. The graph construction step can be done in $O(N)$ time since each element in A' has at most four neighbors. For tree construction (and reconstruction), there are at most $O(\log N)$ levels and at each level, we need to compute the cost of every possible clustering of two neighboring nodes and sort the edges according to these costs. This can be done in $O(N \log N)$. We also need to compute the connectivity information for the next level, but this can again be done in $O(N)$ time since there are at most $O(N)$ edges to look at. Therefore, the total time for tree construction is $O(N \log^2 N)$. The dynamic programming for optimal tree partitioning can be done iteratively in a bottom-up manner, i.e., compute $DP(t, k)$ from $k = 1$ to K, where K is the maximum number of partitions that we will try, for each subtree t starting from the those rooted at a leaf to the whole tree T. The total time for this dynamic programming step is thus $O(K^2 N)$. The solution legalization step can be done in $O(N \log N)$ time. Therefore, the total time complexity of this tree-based approach is $O(N(log^2 N + K^2))$.

4. A DIRECT METHOD

A direct method to solve the NVPP problem is by recursively merging two neighboring cells (or super-cells) in an order of non-decreasing power wastage incurred. Those mergings with less power wastage will be done first. This process can be repeated as long as the power wastage threshold is not exceeded. At the end, we will obtain a set of connected regions without violating the threshold. However, this approach has an intrinsic problem that neighboring cells with the same voltage will first be merged to form big patches. These big patches will forbid some non-neighboring patches of the same or similar voltages to be merged to reduce the number of islands without exceeding the threshold. A simple example to illustrate this situation is shown in fig. 4(a). To tackle this problem, we can coarsen the input $m \times n$ grid to one of a small number of rows and columns ($p \times q$) (fig. 4(b)). Unlike the coarsening step in 3.1, the size of the coarsened grid here should be small (otherwise, the same problem will occur) and it will be done carefully to ensure that the power



(a) Cannot give two islands. (b) Can possibly give two islands after coarsening.

Figure 4: Drawback of the Direct Approach to Solve NVPP.

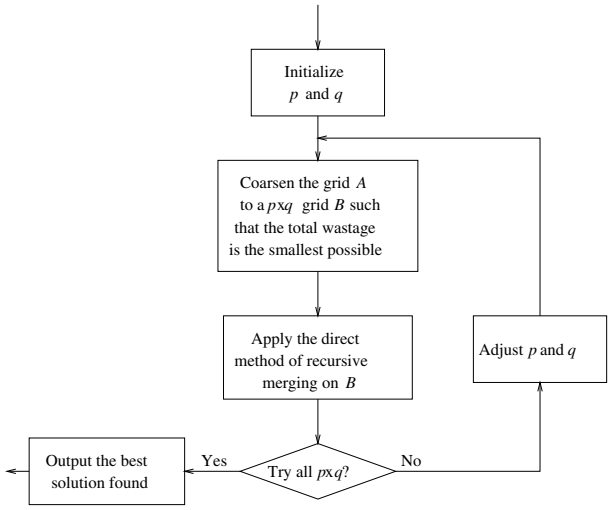


Figure 5: An Overview of the Direct Method.

wastage incurred is the smallest possible. This coarsening problem is a dual version of the *Grid-partitioning Problem* as defined in [6] and we will describe it in more details in the next section. However, it is difficult to figure out the correct values of p and q. If they are too large, the same problem as illustrated in fig. 4(a) will occur, but if they are too small, the power wastage will be a lot. In our implementation, we will try different values of p and q and output the best partitioning. Experimental results have shown that very good results can be obtained by having $p, q \leq 20$ for all the industrial data sets we have tried. Fig. 5 gives an overview of this direct method to solve the NVPP problem.

4.1 Dual Grid-partitioning Problem (DGPP)

In this dual grid-partitioning problem, we are given the original $m \times n$ grid A and two positive integers p and q, we want to partition A into a $p \times q$ grid B where the total power wastage $w(B)$ is as small as possible where $w(B)$ is similarly defined as:

$$w(B) = \sum_{1 \leq i \leq p, 1 \leq j \leq q} w(B[i, j]),$$

the power wastage $w(B[i, j])$ of an element $B[i, j]$ is defined as:

$$w(B[i, j]) = \sum_{(x,y) \in B[i,j]} (\mu(B[i, j]) - A[x, y])$$

and $\mu(B[i, j]) = \max_{(x,y) \in B[i,j]} A[x, y]$ is the maximum power required in $B[i, j]$.

The GPP problem defined in [6] is similar except that a

power wastage threshold ϵ is given and they want to find the smallest possible $p \times q$ such that the threshold is not exceeded. A randomized algorithm with guarantee in solution quality and runtime has been proposed in [5] and is applied to solve the GPP in [6]. For this dual version of GPP, we can similarly apply the technique to solve the problem. In the randomized algorithm, a load is assigned to every row and column of the grid A and an iterative doubling technique is used. Starting with unit load at each separator (row and column) and a uniform $p \times q$ grid, an element in the current $p \times q$ grid is chosen randomly with probability proportional to the amount of power wastage of that element. The larger the power wastage, the higher the chance an element will be chosen. Then all the rows and columns intersecting with the chosen element will have their loads increased by a certain rate. The current grid is then refined according to the new loads by dividing the total load evenly into p rows in the vertical direction and q columns in the horizontal direction. The intuition is that if an element has higher power wastage, the rows and columns intersecting with it are more likely to be chosen as gridlines in the next iteration. This step of refining the grid and updating the load is repeated until the total number of iterations exceeds a certain bound or the total power wastage is less than a threshold ϵ. By binary search on the threshold ϵ, we can find the smallest possible threshold achievable by a $p \times q$ grid. Careful analysis can show that the expected number of iterations is bounded and more details about the algorithm and the analysis can be found in [5].

4.2 Time Complexity

The randomized algorithm to solve the DGPP will be invoked a constant number of times depending on the number of combinations of $p \times q$ that we want to try. In our implementation (as described by fig. 5), this constant is 16 in all our experiments. The runtime of the randomized algorithm is $O((m + n + pq)p \log(mn) \log \frac{K}{\varepsilon})$ where K is the difference between the upper and lower bounds of the binary search on ϵ and ε is the error bound we allow in the search. Since $p, q \leq 20$ in our implementation, the time complexity to solve the DGPP problem once is $O((m + n) \log(mn) \log \frac{K}{\varepsilon})$. The second part of recursively merging elements in the resultant coarsened grid obtained by the randomized algorithm can be done very efficiently in $O(pq)$ time. Again, $p, q \leq 20$ and this is equivalent to constant time. Therefore the total time for this direct method is $O(mn + (m + n) \log(mn) \log \frac{K}{\varepsilon})$. The extra $O(mn)$ time is due to a pre-processing step that allows us to calculate the power wastage of any rectangular region of the grid A in constant time in the randomized algorithm.

5. EXPERIMENTAL RESULTS

In the experiments, we will compare our tree partitioning approach (Tree-Alg) with the approach in [6] (called TS-Alg) and the greedy direct method (G-Alg). We used the same set of industrial designs as in [6], (please refer to [6] for details on how the data sets are prepared) and, similar to [6], the bound on total power increase is computed as a certain percentage of the maximum power increase, which corresponds to the total power increase when the voltages of all the cells are raised to the highest required voltage on the entire chip. All the experiments are conducted using a Dell Optiplex 280 with an Intel P4 3.2GHz CPU and 2GB RAM.

Table 1 compares the output sizes of our Tree-Alg with those of TS-Alg and G-Alg. Here, we used the same power increase bounds as in [6] to order to have a fair comparison. The designs are listed with increasing input sizes, i.e., the size of the array A. Notice that the experiments of [6] are conducted on a machine with 1.95 GHz and 11.7GB memory and their reported run time does not include the $O(m \times n)$ pre-processing time for the GPP. For the number of voltage islands, the best result is bolded on each row. From the table, we can see that our Tree-Alg can give a much smaller number of voltage islands in comparison with TS-Alg (reduced by a factor of 5.1 on average), because the unnecessary restriction on the structure and shapes of the voltage islands is relaxed. It is hard to compare the run time directly because different machines are used and a non-trivial amount of pre-processing time is omitted in the reported run time of [6]. However, we can see from table 1 that the run time of our Tree-Alg is very affordable in practice. In comparison with the greedy direct approach G-Alg, the numbers of voltage islands generated are similar (Tree-Alg gives the same or a smaller number of voltage islands in 8 out of the 10 cases), but Tree-Alg can run a lot much faster. Actually, most of the run time in G-Alg is spent on solving the DGPP (the pre-processing step to coarsen the input grid), and the run time to perform the recursive merging is negligible.

Table 2 displays another comparison between Tree-Alg and TS-Alg in which the number of voltage islands generated is similar and we want to know how much saving in power wastage can be achieved. We can see from column three that the saving is 23% on average, with a runtime of just about 25 seconds on average. Fig. 6-7 show some snapshots of the original placements and the voltage islands generated by Tree-Alg for some of the data sets. In these diagrams, the darker the color, the higher the voltage requirement is. We can see from these figures that Tree-Alg can carefully outline the boundaries of the islands to reduce power wastage. Although the islands formed are rugged in shape, their boundaries can be easily smoothened without causing much increase in the total power consumption.

6. REFERENCES

[1] Power Islands: The Evolving Topology of SoC Power Management. *http://www.us.design-reuse.com/articles/article9150.html.*

[2] J. Buurma and L. Cooke. Low-power Design using Multiple $V_T H$ ASIC Libraries. *http://www.sinavigator.com/Low_Power_Design.pdf.*

[3] J. Hu, Y. Shin, N. Dhanwada, and R. Marculescu. Architecting Voltage Islands in Core-based System-on-a-chip Designs. *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 180–185, 2004.

[4] D. E. Lackey, P. S. Zuchowski, T. R. Bednar, D. W. Stout, S. W. Gould, and J. M. Cohn. Managing Power and Performance for System-on-chip Designs using Voltage Islands. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 195–202, 2002.

[5] S. Muthukrishnan and T. Suel. Approximation Algorithms for Array Partitioning Problem. *J. of Algorithms*, 54:85–104, 2005.

[6] H. Wu, I.-M. Liu, Martin D. F. Wong, and Y. Wang. Post-Placement Voltage Island Generation under Performance Requirement. *Proceedings of the International Conference on Computer-Aided Design*, 2005.

Table 1: Comparisons with TS-Alg and G-Alg

Design			Power	# of VI			Ratio	Run time (s)		
Name	# of Cells	Size of A	Bound	Tree-Alg (a)	G-Alg	TS-Alg (b)	($a:b$)	Tree-Alg	G-Alg	TS-Alg
IndustryB	5926	79×790	60%	7	**4**	18	1:2.6	1	7	58
IndustryC	43677	161×2860	50%	**2**	**2**	19	1:9.5	2	19	125
IndustryG	76406	230×3504	60%	**2**	3	19	1:9.5	3	27	105
IndustryH	243188	694×7852	40%	**4**	6	19	1:4.8	8	44	9
IndustryA	317752	732×8793	55%	**5**	**5**	17	1:3.4	13	52	16
IndustryD	397940	1300×8270	35%	8	**4**	17	1:2.1	20	58	49
IndustryI	342113	1199×8931	35%	**3**	5	15	1:5	8	55	39
IndustryJ	737555	1372×12350	35%	**4**	**4**	17	1:4.3	31	70	32
IndustryE	352060	2144×17159	35%	**7**	9	13	1:1.9	12	111	47
IndustryF	306326	2652×20978	40%	**2**	**2**	15	1:7.5	6	125	5
Average							1:5.1	9.6	57	49

Table 2: Comparisons with TS-Alg on Power Wastage

Design	Power Bound		Reduction	# of VI		Run time (s)	
Name	Tree-Alg	TS-Alg		Tree-Alg	TS-Alg	Tree-Alg	TS-Alg
IndustryB	40%	60%	20%	14	18	1	58
IndustryC	25%	50%	25%	21	19	7	125
IndustryG	35%	60%	25%	14	19	14	105
IndustryH	15%	40%	25%	17	19	16	9
IndustryA	30%	55%	25%	17	17	46	16
IndustryD	15%	35%	20%	16	17	43	49
IndustryI	10%	35%	25%	13	15	22	39
IndustryJ	15%	35%	20%	17	17	58	32
IndustryE	25%	35%	10%	9	13	24	47
IndustryF	10%	40%	30%	13	15	16	5
Average			23%			25	49

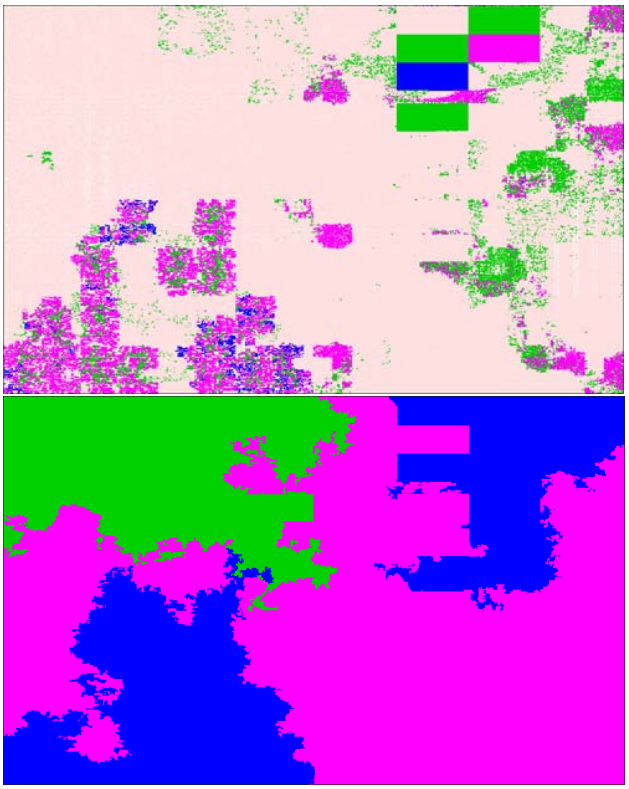


Figure 6: The results of IndustryA (above: original placement; below: 5 voltage islands)

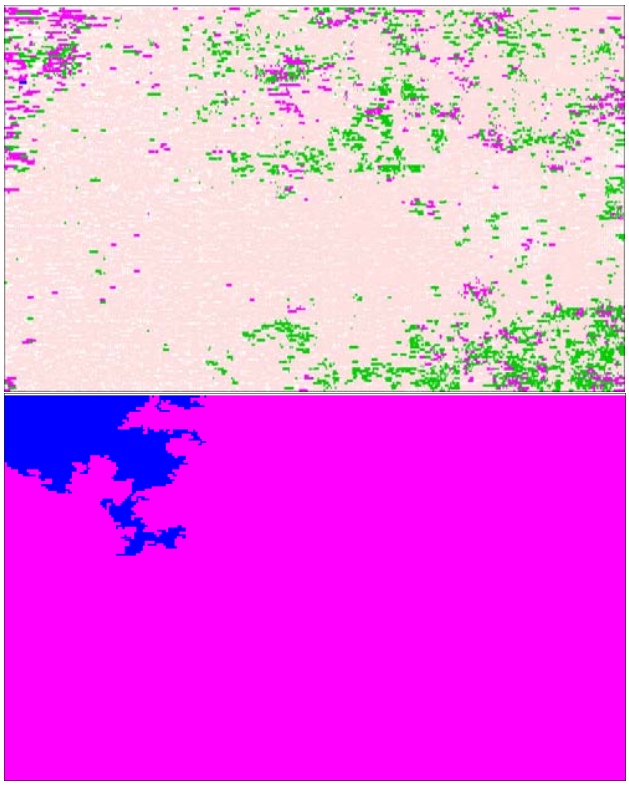


Figure 7: The results of IndustryC (above: original placement; below: 2 voltage islands)