A Clustering- and Probability-based Approach for Time-multiplexed FPGA Partitioning *†

Mango Chia-Tso Chao, Guang-Ming Wu, and Yao-Wen Chang

Department of Computer and Information Science, National Chiao Tung University, Hsinchu 300, Taiwan

Abstract

Improving logic density by time-sharing, time-multiplexed FPGAs (TMFPGAs) have become an important research topic for reconfigurable computing. Due to the precedence and capacity constraints in TMFPGAs, the clustering and partitioning problems for TMFPGAs are different from the traditional ones. In this paper, we propose a two-phase hierarchical approach to solve the partitioning problem for TMFPGAs. With the precedence and capacity considerations for both phases, the first phase clusters nodes to reduce the problem size, and the second phase applies a probability-based iterative-improvement approach to minimize cut cost. Experimental results based on the Xilinx TMFPGA architecture show that our algorithm significantly outperforms previous works.

1 Introduction

Improving logic density by time-sharing, time-multiplexed FPGAs (TMFPGAs) have become an important research topic for reconfigurable computing. In TMFPGAs, a virtual large design is partitioned into multiple stages (or partitions) to share the same smaller physical device than that occupied by traditional FPGAs. Several different architectures have been proposed, such as the Xilinx model [10], Dharma [1], etc. All these models allow dynamic reuse of logic blocks and wire segments by using more than one on-chip SRAM bit to control them. The configurations of logic blocks and wire segments can be changed by reading different SRAM bits.

Figure 1 shows the Xilinx TMFPGA configuration model [10]. The TMFPGA emulates a single circuit design in the sequencing of multiple configurations. In each micro-cycle, the TMFPGA reads in the circuit information from a corresponding configuration SRAM, and then the configurable logic blocks (CLBs) in the TMFPGA are reused to evaluate logic. A user cycle is a cycle passing through all micro-cycles. Each CLB contains micro registers to hold the CLB result. Micro registers hold the intermediate values of combinational logic for later micro-cycles in the same user cycle and reserve the status of flip-flops for the next user cycle. In Xilinx TMFPGAs, there are eight

*This work was partially supported by the National Science Council of Taiwan ROC under Grant No. NSC-88-2215-E-009-064. E-mail: {gis87530, gis85815, ywchang}@cis.nctu.edu.tw

†The preliminary version of this paper was presented on the 1999 International Conference of Computer Aided Design.

micro-cycles in a user cycle. A new configuration is loaded into active configuration memory after all CLB results in the last micro-cycle have been saved.

The objective of the TMFPGA partitioning problem is to minimize the interconnection (the number of micro registers required) between micro-cycles. Unlike a traditional FPGA, the execution order of nodes in a TMFPGA must follow their precedence constraints. For example, a node must be excuted no later than all of its outputs in a combinational circuit. It implies that a cut in a TMFPGA partitioning should be a uni-directional cut. For the TMFPGA partitioning problem, several heuristics such as list scheduling [2, 3] and network-flow-based approach [9] on different architectures were proposed. The network-flow-based approach [9] gives the best results among the previous works for the partitioning problem. The network-flow-based approach first finds a min cut. If the min cut is not at the balanced point, it will randomly move nodes to meet the balance constraint. Thus the optimality may deviate away after nodes are adjusted. In this paper, we propose a two-phase approach, the CPAT method (*C*lustering and *P*robability-based *A*lgorithm for *T*MFPGA), to solve the TMFPGA partitioning problem. The first phase reduces the problem size using a clustering method; the second phase minimizes the interconnection by a probability-based iterative-improvement [7, 8] method. For the first phase, we extend the method used in [4] which is effective in clustering traditional circuits, but may generate a cluster of size exceeding the capacity of a stage in the TMFPGA partitioning. Our solution to the capacity overflow problem is based on a rooted-tree subset-sum formulation; we prove that the rooted-tree subset-sum problem is NP-complete and present an exact exponential-time and a *fully* polynomial-time approximation algorithms [5] for the problem. For the second phase, the probability-based method incroporates the precedence constraints into the 2nd-order probability estimation [6]. Thus, the probability-based method finds the potentially maximum gain among movable nodes. Our method, thus, can globally monitor the changes and can avoid the drawback of the network-flow-based approach. Experimental results, based on the Xilinx TMFPGA architecture [10] with eight micro-cycles (stages), show that our algorithm reduces the maximum numbers of micro registers required by average improvements of 33.2% and 12.4% compared with the List scheduling and the network-flow-based [9] methods.

2 Problem Formulation

We follow the formulation and notation used in [9]. A circuit in a TMFPGA can be represented by a directed hypergraph $G(V, N)$, where V is the set of nodes and N is the set of nets in the circuit. There are two types of nodes in V: combinational nodes (*C-nodes*) and flip-flop nodes (*FF-nodes*). Each node $v \in V$ has a weight $w(v)$. The weight of a set U ($U \subseteq V$), $W(U)$, is given by $\sum_{v \in U} w(v)$. For a net $n = \{v_1, v_2, \ldots, v_p\}$ with p nodes, let v_1 be the *fan-out node* whose output signal is the input signal to $v_j \in n$ ($2 \leq j \leq p$), and let $v_j \in n$ ($2 \leq j \leq p$) be the *fan-in node* whose input signal is the output signal from v_1.

To fit into a TMFPGA, a circuit is partitioned into k stages, such that the logic blocks and wire segments in

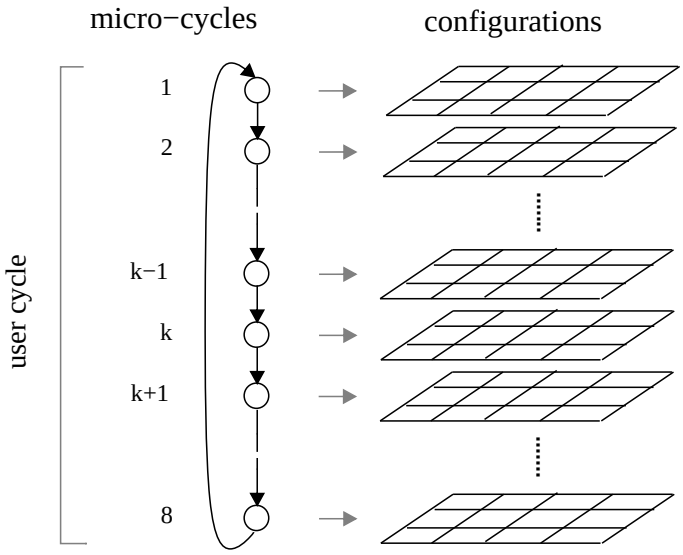


Figure 1: The Xilinx time-multiplexed FPGA configuration model.

different stages can share the same physical TMFPGA device. These k stages form one user cycle, and one user cycle should produce the same results on the outputs as would be seen by a non-time-multiplexed device. In order to ensure the correct results produced in a user cycle, every nodes must be evaluated in a proper order. According to the Xilinx architecture [10], the following three precedence constraints must be satisfied:

1. Each combinational node (C-node) must be scheduled in a stage no later than all its output nodes.
2. Each flip-flop node (FF-node) must be scheduled in a stage no earlier than all its input C-nodes. This rule guarantees that flip-flop input values are calculated before they are stored.
3. Each FF-node must be scheduled in a stage no earlier than all its output nodes. This rule guarantees that all the nodes that use the value of the flip-flop use the same value: the value of flip-flop from the previous user cycle.

The above constraints define a partial temporal ordering on the nodes in the circuit. Let $Pre(v)$ be the precedence of a node v. For two nodes u and v, let $Pre(u) \preceq Pre(v)$ denote that node u must be scheduled no later than node v. In other words, for a net $n = \{v_1, v_2, \ldots, v_p\}$, where v_1 is the fan-out node and v_j, $2 \leq j \leq p$, is the fan-in node.

- if v_1 is a C-node, then $Pre(v_1) \preceq Pre(v_j)$ for $2 \leq j \leq p$;
- if v_1 is an FF-node, then $Pre(v_j) \preceq Pre(v_1)$ for $2 \leq j \leq p$.

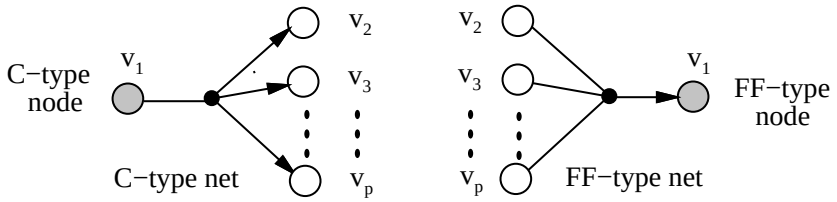


Figure 2: Precedence constraints. Shaded nodes and white nodes represent the fan-out nodes and fan-in nodes, respectively.

By the two constraints, we can decide the directions of nets in the graph and classify nets into two types: a net is *C-type* if its v_1 is a C-node, and a net is *FF-type* if its v_1 is an FF-node, as shown in Figure 2.

In TMFPGAs, micro registers are required between stages to store the data of nodes for use in later micro-cycles. Let $Cut(a, b)$ be the set of micro registers between stage a and stage b. A k-stage TMFPGA contains k cuts, $Cut(1, 2)$, $Cut(2, 3)$,..., $Cut(k-1, k)$, and $Cut(k, 1)$. For a C-type net, the data of its fan-out node must be held until the last stage containing a fan-in node of the net. For an FF-type net, the data of its fan-out node must be held not only in the rest stages of the current user cycle but also from the first stage to the last stage of all its fan-in nodes in the next user cycle. For a net n =$\{v_1, v_2, \ldots, v_p\}$, let $s(v) = j$ if $v \in V_j$, $\alpha(n)$ denote the number of micro registers used in net n, and k denote the number of stages. $\alpha(n)$ is given as follows.

- $\alpha(n) = \max\{s(v_j)|2 \leq j \leq p\} - s(v_1)$, if net n is C-type.
- $\alpha(n) = k - s(v_1) + \max\{s(v_j)|2 \leq j \leq p\}$, if net n is FF-type.

Figure 3 shows the registers needed in a net for a 4-stage TMFPGA. In Figure 3(a), the data of a C-type fan-out node is held from stage 1, the stage of the fan-out node, to stage 3, the last stage of the fan-in nodes. It uses three micro registers, one for $Cut(1, 2)$ and $Cut(2, 3)$ each. In Figure 3(b), the data of an FF-type fan-out node is held from stage 3, the stage of fan-out node, to stage 4, then back to stage 1 of next user cycle and finally to stage 2, the last stage of fan-in nodes. It uses three registers, one for $Cut(3, 4)$, $Cut(4, 1)$, and $Cut(1, 2)$ each.

The *k-stage TMFPGA partitioning* problem is to partition a circuit $G(V, N)$ into k non-overlapping subsets V_1,V_2,...,V_k, such that the maximum interconnection (the number of micro registers) between each two adjacent stages is minimized, and the following properties are satisfied:

(1) $\bigcup_{i=1}^{k} V_i = V$.

(2) *Precedence constraint*: Let $s(v) = j$ if $v \in V_j$. For each two nodes u and v, if $Pre(u) \preceq Pre(v)$, then $s(u) \leq s(v)$.

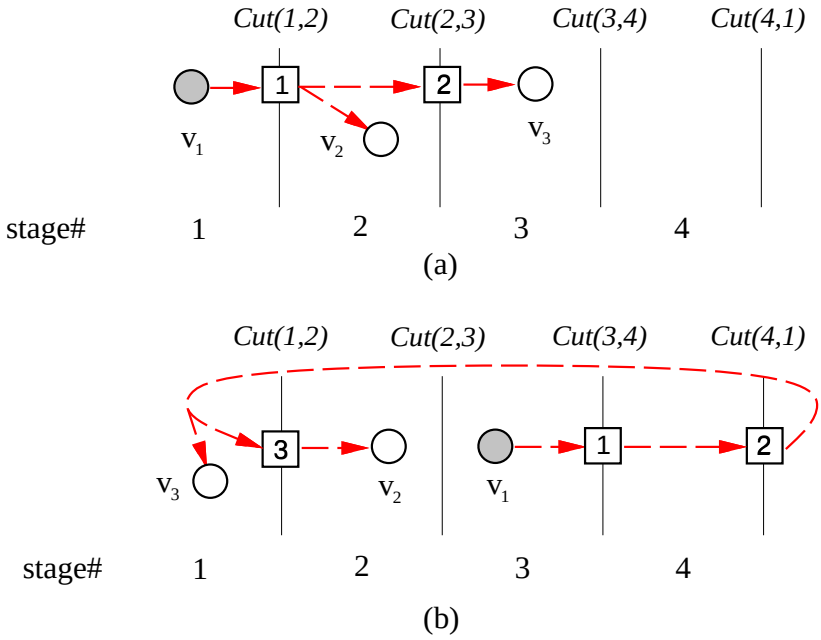


Figure 3: (a) Two micro registers, indicated by □, used in a C-type net $\{v_1, v_2, v_3\}$; (b) Three micro registers used in an FF-type net $\{v_1, v_2, v_3\}$.

(3) *Balance constraint*: For each subset V_i, $W(V_i)$ is bounded by a factor r as follows:

$$\frac{W(V)}{k}(1-r) \leq W(V_i) \leq \frac{W(V)}{k}(1+r), 0 \leq r \leq 1.$$

(4) *Timing constraint*: Let D be the length of the longest path in a circuit. The length of the longest path in each stage is upper bounded by $\lceil \frac{D}{k} \rceil$.

3 The Two-Phase CPAT Algorithm

The k-stage TMFPGA partitioning problem can be handled by repeatedly solving $k-1$ TMFPGA bipartitioning problems. We shall focus our discussions on the approach for solving the TMFPGA bipartitioning problem. Our solution to this problem is based on a two-phase hierarchical approach: clustering followed by a probability-based iterative-improvement formulation.

3.1 Phase I: The Clustering Algorithm

An effective clustering algorithm can greatly improve the quality of the precedence-constrained partitioning results and speed up the later partitioning algorithm by reducing the problem size. The MFFS algorithm is effective in clustering traditional circuits [4], but may generate a cluster of size larger than the capacity of a stage in the TMFPGA partitioning.

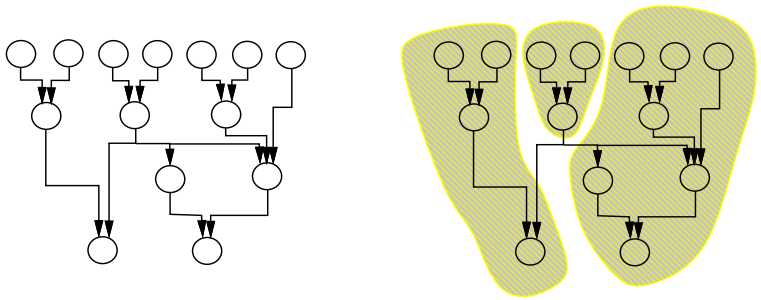
In the following, we propose a clustering method based on the MFFS, which can control the size of a cluster. The definitions of FFS and MFFS are described as follows. For a given node v in a circuit,

- FFS_v = $\{u \mid$ *every path from u to some primary output passes through v in the circuit*$\}$.
- $MFFS_v$ = $\{u \mid$ *for all* FFS_v, $u \in FFS_v\}$.

A circuit can be represented in the TMFPGA by a directed graph. For a given circuit C_i and a node v, an MFFS cluster rooted at v can be obtained by using the following procedure:

- Convert C_i to a directed graph, $G(V, N)$, where V is a set of nodes which corresponds to C_i, and N is a set of directed edges. A directed edge (i, j) exists if node j is a fan-in of node i.
- Cut all the fan-out edges of the root node v; search all other nodes in graph $G(V, N)$ starting from the primary outputs of the C_i. The nodes in $G(V, N)$ that were not traversed belong to the $MFFS_v$.

The MFFS construction algorithm described above is used to obtain one MFFS cluster. To cluster the entire circuit, we need to apply the MFFS construction algorithm repeatedly. The MFFS clustering algorithm works as follows: For a given circuit C_i, let $Roots$ = {all primary outputs in C_i}. Then, extract a node $v \in Roots$ and use the MFFS construction algorithm to construct $MFFS_v$. This process is repeated until $Roots$ is empty. Then remove all currently constructed MFFS clusters from C_i, resulting in a reduced circuit C_i' whose primary outputs are input nodes to the removed MFFS clusters. Repeat the same procedure for the new circuit C_i' recursively until all nodes in C_i are grouped into MFFS clusters. For example, the circuit depicted in Figure 4(a) can be clustered into three clusters (see Figure 4(b)).



(a) (b)

Figure 4: (a) The original circuit; (b) Clustering by MFFS clustering.

We present in the following two algorithms to handle a cluster of size exceeding the capacity of a stage in the TMFPGA partitioning. Our method decomposes a cluster C_i (rooted at v) according to the two cases: (1) C_i is a rooted tree, and (2) C_i is an acyclic graph. Our target is to partition C_i into two balanced sets with the minimal cut size.

We first consider the case where a circuit C_i is a rooted tree. Let T_{v_i} denote the subtree rooted at v_i, where $v_i \in C_i$. For nodes $v_1, v_2, \ldots$, and v_d in respective $T_{v_1}, T_{v_2}, \ldots$, and T_{v_d}, let $\kappa(v_1, v_2, \ldots, v_d)$ denote the total

weights of nodes in $T_{v_1}, T_{v_2}, \ldots$, and T_{v_d}. We define an *element* $x = (\kappa(v_1, v_2, \ldots, v_d), T_{v_1}, T_{v_2}, \ldots, T_{v_d})$, where $v_1, v_2, \ldots, v_d$ represent the respective roots of disjoint subtrees $T_{v_1}, T_{v_2}, \ldots, T_{v_d}$. For an element $x = (\kappa(v_1, v_2, \ldots, v_d), T_{v_1}, T_{v_2}, \ldots, T_{v_d})$, let $|x| = d$ and $\pi(x) = \kappa(v_1, v_2, \ldots, v_d)$. An element y is called a *singleton element* if it contains only one subtree. For an element $x_i = (\kappa(v_{i,1}, v_{i,2}, \ldots, v_{i,d}), T_{v_{i,1}}, T_{v_{i,2}}, \ldots, T_{v_{i,d}})$ and a singleton element $y_j = (\kappa(v_j), T_{v_j})$, if $T_{v_j} \not\subset T_{v_{i,l}}$, $1 \leq l \leq d$, let $x_i \uplus y_j = (\kappa(v_{i,1}, v_{i,2}, \ldots, v_{i,d}, v_j), T_{v_{i,1}}, T_{v_{i,2}}, \ldots, T_{v_{i,d}}, T_{v_j})$; if $T_{v_j} \subset T_{v_{i,l}}$, $1 \leq l \leq d$, let $x_i \uplus y_j = (\kappa(\hat{V}), \hat{T})$, where the set $\hat{V} = \{v_{i,1}, v_{i,2}, \ldots, v_{i,d}, v_j\} - \{v_{i,l}\}$ and $\hat{T} = \{T_{v_{i,1}}, T_{v_{i,2}}, \ldots, T_{v_{i,d}}, T_{v_j}\} - \{T_{v_{i,l}}\}$. Let h denote a half of the total number of nodes in C_i. The rooted-tree subset-sum problem is to cut C_i into at most q subtrees such that the total weights of nodes in the subtrees is equal to h. We formulate the rooted-tree subset-sum problem as follows.

- **The Rooted-Tree Subset-Sum problem:** Given a set R of singleton elements associated with a rooted tree C_i and two integers q and h, find an element x derived by a sequence of $\uplus$ operations such that $\pi(x) = h$ and $|x| \leq q$.

Theorem 1 *The decision problem of the Rooted-Tree Subset-Sum problem is NP-complete.*

Proof: We first show that Rooted-Tree Subset-Sum problem is in NP. Given a set R associated with a rooted tree and two integers q and h, we let the subset R' of R be the certificate. Checking whether $h = \pi(\uplus_{x \in R'} x)$ and $|\uplus_{x \in R'} x| = q$ can be accomplished by a verification algorithm in polynomial time.

The SUBSET-SUM problem is an NP-complete problem. [5] We now show that SUBSET-SUM $\leq_P$ Rooted-Tree Subset-Sum. Given an instance $< \hat{S}, t >$ of the subset-sum problem, the reduction algorithm constructs a tree (a circuit) C of the Rooted-Tree Subset-Sum problem such that there exists a subset in $\hat{S}$ whose sum is equal to t if and only if there exists an element x associated with C, where $\pi(x) = t$.

The heart of the reduction is a tree representation of $\hat{S}$. Let $\hat{S} = \{s_1, s_2, \ldots, s_i\}$ be a set consisting of i integers. We construct the tree $C(V, N)$ with $i + 1$ nodes associated with $\hat{S}$ as follows:

- Add a root v_0 with weight ∞ to V.
- For each integer $s_j \in S$, add a node v_j with weight s_j to V and a directed edge (v_0, v_j) to N.

Every subtree of C except T_{v_0} has only one node and is disjoint to each other. We have $R = \{(\kappa(v_0), T_{v_0}), (\kappa(v_1), T_{v_1}), \ldots, (\kappa(v_i), T_{v_i})\}$ associated with $C(V, N)$. Let q equal i, $\hat{S}' \subseteq \hat{S}$ such that $t = \Sigma_{s_j \in \hat{S}'} s_j$, and $y_k = (\kappa(v_k), T_{v_k})$. Then we find the element $x = \uplus y_j$, where y_j is associated with $s_j \in \hat{S}'$, such that $\pi(x) = t$ and $|x| \leq q$.

Conversely, suppose that there exists an element $x = (\kappa(v_1, v_2, \ldots, v_d), T_{v_1}, T_{v_2}, \ldots, T_{v_d})$. Let $|x|$ equal d and $\pi(x)$ equal $\kappa(v_1, v_2, \ldots, v_d)$ such that $\pi(x) = t$. Then, the sum of the subset $\{v_{j1}, v_{j2}, \ldots, v_{jk}\}$ is equal to t. $\square$

We give an exponential-time exact algorithm as well as a fully polynomial-time approximation scheme [5] for the Rooted-Tree Subset-Sum problem, listed in Figures 6 and 7, respectively. For a sequence $L =< (\kappa(v_{1,1}, \ldots, v_{1,i_1}), T_{v_{1,1}}, \ldots, T_{v_{1,i_1}}), (\kappa(v_{2,1}, \ldots, v_{2,i_2}), T_{v_{2,1}}, \ldots, T_{v_{2,i_2}}), \ldots, (\kappa(v_{m,1}, \ldots, v_{m,i_m}), T_{v_{m,1}}, \ldots, T_{v_{m,i_m}}) >$ and $(\kappa(v_j), T_{v_j})$, let $L + (\kappa(v_j), T_{v_j})$ denote the sequence derived from a series of $\uplus$ operations on each element of L with the singleton element $(\kappa(v_j), T_{v_j})$. For example, if $L =< (1, T_{v_1}), (3, T_{v_2}), (5, T_{v_3}), (6, T_{v_4}) >$, then $L + (2, T_{v_5}) =< (3, T_{v_1}, T_{v_5}), (5, T_{v_2}, T_{v_5}), (7, T_{v_3}, T_{v_5}), (8, T_{v_4}, T_{v_5}) >$ (if $T_{v_1}, \ldots, T_{v_5}$ do not share any node).

We use an auxiliary procedure Merge-Lists(L, L') that returns the sorted list by merging its two sorted input lists L and L', and remove the duplicate elements. The polynomial-time approximation algorithm Approx-Rooted-Tree-Subset-Sum is performed by $trimming$ each list L_i after an $\uplus$ operation. We use a trimming parameter ϵ such that $0 \leq \epsilon \leq 1$. To $trim$ a list L by ϵ means to remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for each element y removed from L, there exists an element z still in L', where $(1 - \epsilon)\pi(y) \leq \pi(z) \leq \pi(y)$. Line 3 initializes the list L_0 to be the list containing just the element $(0, \emptyset)$. Lines 4–6 perform the $\uplus$ operation and trimming operations in a topological order. Lines 7 and 8 remove each element x, $\pi(x) > h$ and $|x| > q$. We can show that Approx-Rooted-Tree-Subset-Sum listed in Figure 7 runs in time polynomially in both $|R|$ and $1/\epsilon$; i.e., it is a fully polynomial-time approximation scheme [5].

We give an example of Approx-Rooted-Tree-Subset-Sum in the following. Suppose we have a list of singleton elements

$$L =< (8, T_{v_1}), (3, T_{v_2}), (4, T_{v_3}), (1, T_{v_4}), (1, T_{v_5}), (2, T_{v_6}), (1, T_{v_7}), (1, T_{v_8}) >$$

associated with the rooted-tree in Figure 5, in which the weight of each vertex is equal to 1. The target is to find an element x, where $\pi(x) = h = 4$ and $|x| = q = 1$ with $\epsilon = 0.2$. The trimming parameter ρ is $\epsilon/8 = 0.025$. The Approx-Rooted-Tree-Subset-Sum computes the elements as follows:

line 2: $L_0 =$ $< (0, \emptyset) >$,
line 4: pick $(8, T_{v_1})$,
line 5: $L_1 =$ $< (0, \emptyset), (8, T_{v_1}) >$,
line 6: $L_1 =$ $< (0, \emptyset), (8, T_{v_1}) >$,
line 7: $L_1 =$ $< (0, \emptyset) >$,
line 8: $L_1 =$ $< (0, \emptyset) >$,

line 4: pick $(3, T_{v_2})$,
line 5: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}) >$,
line 6: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}) >$,
line 7: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}) >$,
line 8: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}) >$,

line 4: pick $(4, T_{v_3})$,
line 5: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}), (4, T_{v_3}), (7, T_{v_2}, T_{v_3}) >$,
line 6: $L_2 =$ $< (0, \emptyset), (3, T_{v_2}), (4, T_{v_3}), (7, T_{v_2}, T_{v_3}) >$,

line 7: $L_2 = < (0,\emptyset), (3,T_{v_2}), (4,T_{v_3}) >,$
line 8: $L_2 = < (0,\emptyset), (3,T_{v_2}), (4,T_{v_3}) >,$

line 4: pick $(1,T_{v_4})$,
line 5: $L_2 = < (0,\emptyset), (1,T_{v_4}), (3,T_{v_2}), (4,T_{v_3}), (5,T_{v_3},T_{v_4}) >,$
line 6: $L_2 = < (0,\emptyset), (1,T_{v_4}), (3,T_{v_2}), (4,T_{v_3}), (5,T_{v_3},T_{v_4}) >,$
line 7: $L_2 = < (0,\emptyset), (1,T_{v_4}), (3,T_{v_2}), (4,T_{v_3}) >,$
line 8: $L_2 = < (0,\emptyset), (1,T_{v_4}), (3,T_{v_2}), (4,T_{v_3}) >,$

line 4: pick $(1,T_{v_5})$,
line 5: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_4},T_{v_5}), (3,T_{v_2}), (4,T_{v_3}), (5,T_{v_3},T_{v_3}) >,$
line 6: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_4},T_{v_5}), (3,T_{v_2}), (4,T_{v_3}), (5,T_{v_3},T_{v_3}) >,$
line 7: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_4},T_{v_5}), (3,T_{v_2}), (4,T_{v_3}) >,$
line 8: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (3,T_{v_2}), (4,T_{v_3}) >,$

line 4: pick $(2,T_{v_6})$,
line 5: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_6}), (3,T_{v_2}), (3,T_{v_4},T_{v_6}), (3,T_{v_5},T_{v_6}), (4,T_{v_3}), (5,T_{v_2},T_{v_6}) >,$
line 6: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_6}), (3,T_{v_2}), (3,T_{v_4},T_{v_6}), (3,T_{v_5},T_{v_6}), (4,T_{v_3}), (5,T_{v_2},T_{v_6}) >,$
line 7: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_6}), (3,T_{v_2}), (3,T_{v_4},T_{v_6}), (3,T_{v_5},T_{v_6}), (4,T_{v_3}) >,$
line 8: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (2,T_{v_6}), (3,T_{v_2}), (4,T_{v_3}) >,$

line 4: pick $(1,T_{v_7})$,
line 5: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (2,T_{v_6}), (2,T_{v_4},T_{v_7}), (2,T_{v_5},T_{v_7}), (3,T_{v_2}), (3,T_{v_6},T_{v_7}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_7}) >,$
line 6: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (2,T_{v_6}), (2,T_{v_4},T_{v_7}), (2,T_{v_5},T_{v_7}), (3,T_{v_2}), (3,T_{v_6},T_{v_7}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_7}) >,$
line 7: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (2,T_{v_6}), (2,T_{v_4},T_{v_7}), (2,T_{v_5},T_{v_7}), (3,T_{v_2}), (3,T_{v_6},T_{v_7}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_7}) >,$
line 8: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (2,T_{v_6}), (3,T_{v_2}), (4,T_{v_3}) >,$

line 4: pick $(1,T_{v_8})$,
line 5: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (1,T_{v_8}), (2,T_{v_6}), (2,T_{v_4},T_{v_8}), (2,T_{v_5},T_{v_8}), (2,T_{v_7},T_{v_8}), (3,T_{v_2}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_8}) >,$
line 6: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (1,T_{v_8}), (2,T_{v_6}), (2,T_{v_4},T_{v_8}), (2,T_{v_5},T_{v_8}), (2,T_{v_7},T_{v_8}), (3,T_{v_2}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_8}) >,$
line 7: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (1,T_{v_8}), (2,T_{v_6}), (2,T_{v_4},T_{v_8}), (2,T_{v_5},T_{v_8}), (2,T_{v_7},T_{v_8}), (3,T_{v_2}), (4,T_{v_3}),$
$(4,T_{v_2},T_{v_8}) >,$
line 8: $L_2 = < (0,\emptyset), (1,T_{v_4}), (1,T_{v_5}), (1,T_{v_7}), (1,T_{v_8}), (2,T_{v_6}), (3,T_{v_2}), (4,T_{v_3}) > .$

The algorithm returns $(4,T_{v_3})$, where $\pi(4,T_{v_3}) = 4$, which is bounded in $\epsilon = 20\%$ of the optimal answer.

Theorem 2 *Approx-Rooted-Tree-Subset-Sum is a fully polynomial-time approximation scheme for the Rooted-Tree Subset-Sum Problem.*

Proof: In line 6, the trimming procedure removes each element y where $\pi(y)$ is greater than t from L_i. The rest elements of L_i are generated by selecting a subset of R and applying a sequence of $\uplus$ operations on the selected elements. Therefore, the element x^* returned in line 9 is indeed derived from a subset of R. It remains to show that the $\pi(x^*)$ is not smaller than $1-\epsilon$ times an optimal solution, and we must also show that the algorithm runs

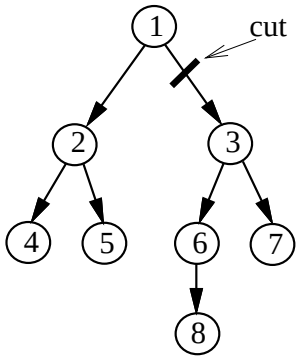


Figure 5: A rooted-tree with eight vertices. The tree has a minimum cut (cut-size = 1) which partitions the tree into two balanced parts.

in polynomial time.

To show that the relative error of the returned answer is small, note that when list L_i is trimmed, we introduce a relative error of at most ϵ/l between the representative π values of the elements remaining and the π values of the elements before trimming. By induction on i, it can be shown that for each possible element y in L_i produced by the Exact-Rooted-Tree-Subset-Sum algorithm, there exists an element $x \in L_i$ produced by the Approx-Rooted-Tree-Subset-Sum algorithm such that

$$(1 - \epsilon/l)^i \pi(y) \leq \pi(x) \leq \pi(y). \tag{1}$$

If y^* denotes an optimal solution to the Rooted-Tree Subset-Sum problem, then there is a $x^* \in L_l$ such that

$$(1 - \epsilon/l)^l \pi(y^*) \leq \pi(x^*) \leq \pi(y^*), \tag{2}$$

where $\pi(x^*)$ is the π value of the element x^* returned by Approx-Rooted-Tree-Subset-Sum. Since $l \geq 1 > \epsilon$, it can be shown that

$$\frac{d}{dl}(1 - \frac{\epsilon}{l})^l > 0. \tag{3}$$

It implies that the function $(1 - \epsilon/l)^l$ increases with l, so that $l > 1$ implies

$$1 - \epsilon < (1 - \epsilon/l)^l, \tag{4}$$

and thus,

$$(1 - \epsilon)\pi(y^*) \leq \pi(x^*). \tag{5}$$

Therefore, the π value of x^* returned by Approx-Rooted-Tree-Subset-Sum is not smaller than $1 - \epsilon$ times the π value of the optimal solution y^*.

To show that this is a fully polynomial-time approximation scheme, we derive a bound on the length of L_i. After trimming, successive elements x and x' of L_i must have the relationship $\pi(x)/\pi(x') > 1/(1-\epsilon/l)$. That is, their π values must differ by a factor of at least $(1-\epsilon/l)$. Therefore, the number of elements in each L_i is at most

$$\log_{1/(1-\epsilon/l)} h = \frac{\ln h}{-\ln(1-\epsilon/l)} \leq \frac{l \ln h}{\epsilon} \tag{6}$$

since $\ln(1+i) \leq i$ for $i > -1$. This bound is polynomial in the number l of the given input elements, in the number of bits $\ln h$ needed to represent h, and in $1/\epsilon$. Since the running time of Approx-Rooted-Tree-Subset-Sum is polynomial in the length of L_i, Approx-Rooted-Tree-Subset-Sum is a fully polynomial-time approximation scheme. □

Algorithm: Exact-Rooted-Tree-Subset-Sum(R, h, q)
Input: R—a set associated with a rooted tree $C_i(V, N)$.
h: a target integer.
q: an upper bound.
Output: x^*—an element s.t. $\pi(x^*)$ is as large as possible but not large than h and $|x^*| \leq q$.
1. $l \leftarrow |R|$;
2. $L_0 \leftarrow < (0, \emptyset) >$;
3. **for** $i \leftarrow 1$ *to* l **do**
4. Pick $(\kappa(v_j), T_{v_j}) \in R$ according to the topological order of node v_j in C;
5. $L_i \leftarrow$ Merge-Lists($L_{i-1}, L_{i-1} + (\kappa(v_j), T_{v_j})$);
6. Remove the element $(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})$ from L_i s.t. $\kappa(v_{i,1}, \ldots, v_{i,d})$ is greater than h;
7. Remove the element $(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})$ from L_i s.t. $|(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})|$ is greater than q;
8. **Output** the element x^* in L_l s.t. $\pi(x^*)$ is the largest.

Figure 6: The exact algorithm for the Rooted-Tree Subset-Sum problem.

Approx-Rooted-Tree-Subset-Sum tells us how to partition a rooted-tree circuit. If its results contain infeasible trees, we need to apply Approx-Rooted-Tree-Subset-Sum repeatedly.

For the case where C_i (rooted at v) is an acyclic graph. We can perform breadth-first search from node v and obtain a rooted tree, and then apply Approx-Rooted-Tree-Subset-Sum on the tree.

3.2 Phase II: The Probability-Based Algorithm

The probability-based iterative-improvement method extends the work [6] to fit the architecture of Xilinx TMF-PGAs.

Algorithm: Approx-Rooted-Tree-Subset-Sum(R, h, q, ϵ)
Input: R—a set associated with a rooted tree $C_i(V, N)$.
h: a target integer.
q: an upper bound.
Output: x^*—an element s.t. $\pi(x^*)$ is as large as possible but not large than h and $|x^*| \leq q$.
1. $l \leftarrow |R|$;
2. $L_0 \leftarrow < (0, \emptyset) >$;
3. **for** $i \leftarrow 1$ to l **do**
4. Pick $(\kappa(v_j), T_{v_j}) \in R$ according to the topological order of node v_j in C;
5. $L_i \leftarrow$ Merge-Lists($L_{i-1}, L_{i-1} + (\kappa(v_j), T_{v_j})$);
6. $L_i \leftarrow$ Trim($L_i, \epsilon/l$);
7. Remove the element $(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})$ from L_i s.t. $\kappa(v_{i,1}, \ldots, v_{i,d})$ is greater than h;
8. Remove the element $(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})$ from L_i s.t. $|(\kappa(v_{i,1}, \ldots, v_{i,d}), T_{v_{i,1}}, \ldots, T_{v_{i,d}})|$ is greater than q;
9. **Output** the element x^* in L_l s.t. $\pi(x^*)$ is the largest.

Subroutine: Trim(L_i, ϵ)
1. $m \leftarrow |L_i|$;
2. $L' \leftarrow < y_1 >$, where y_1 is the first element in L_i;
3. $last \leftarrow \pi(y_1)$;
4. **for** $i \leftarrow 2$ to m **do**
5. **if** $last < (1 - \epsilon)\pi(y_i)$ **then**
6. append y_i onto the end of L';
7. $last \leftarrow \pi(y_i)$;
8. **Output** L'

Figure 7: The fully polynomial-time approximation scheme for the Rooted-Tree Subset-Sum problem.

3.2.1 Iterative-Improvement Approach

In the TMFPGA bipartitioning problem, the set V of nodes is divided into two subsets V_1 and V_2, which represent nodes in two stages. For any two nodes u, v in V, if $Pre(u) \preceq Pre(v)$, then u, v are in the same stage, or u is V_1 and v is in V_2. Further, V_1 and V_2 must satisfy the balance constraint. The size of $Cut(2,1)$ equals the number of total registers in the circuit, which cannot be reduced any more. Therefore, we only need to minimize the size of $Cut(1,2)$ in the TMFPGA bipartitioning problem. In the second step of CPAT, we present the PAT (*P*robability-based *A*lgorithm for *T*MFPGA), which applies a probability-based, iterative-improvement approach to minimize the size of $Cut(1,2)$. (Figure 10 summarizes PAT.) We first use the topological sort to obtain an initial partitioning that satisfies the balance and the precedence constraints (line 1 in Figure 10). During the iterative improvement, each node is assigned a gain, representing the benefit of moving the node to the other subset. In each pass (lines 4–18 in Figure 10), we choose a node with the largest gain and check if it will violate the balance or the precedence constraint after moving the node. If it is feasible to move the node, it is temporarily moved and locked. Select the best sequence of moves and make them permanent. Repeat the above process in a pass until no better cutsize is found.

3.2.2 The Precedence Constraint

Because of the precedence constraint, moving a node to the other subset may not be valid. We use the following two rules to judge if a node can be moved:

R1: A node v in V_1 can be moved if all its successors in V_1 have been moved.

R2: A node v in V_2 can be moved if all its ancestors in V_2 have been moved.

For example, in Figure 8(a), v_2 cannot be moved according to Rule R1. In Figure 8(b), v_3 cannot be moved according to Rule R2.

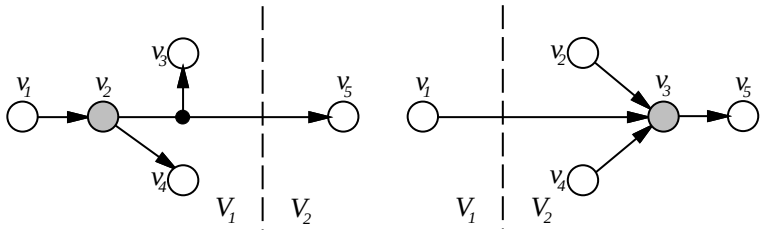


Figure 8: The precedence constraints. Shaded nodes cannot be moved to the other stage due to the precedence constraints.

After a node v is moved to the other stage, some of its neighbors may also be blocked in that stage due to the precedence constraint. We use the following two rules to determine whether such neighbors should be blocked

(see line 14 in Figure 10):

R3: If v is moved from V_1 to V_2, all its successors should be blocked in V_2.

R4: If v is moved from V_2 to V_1, all its ancestors should be blocked in V_1.

3.2.3 Gains of Nodes

In the PAT, each node is given a probability for moving it to the other set. Based on these probabilities, an expected gain of moving a node to the other subset can be evaluated. Before detailing how to compute gains, we shall introduce some notation first.

- $Cutset$: set of nets which need micro registers in $Cut(1,2)$. In other words, a C-type net is not in $Cutset$ if all its nodes are in V_1 or V_2; an FF-type net is not in $Cutset$ only if its fan-out node is in V_2 and all its fan-in nodes are in V_1.
- $c(n)$: cost of net n.
- $p(u)$: probability of moving node u to the other stage.
- $n_i^{1\to2}$: event that net n_i is removed from $Cutset$ by moving nodes to V_2. For a C-type net n_i in $Cutset$, $n_i^{1\to2}$ means all its nodes in V_1 are moved to V_2; for an FF-type net n_i in $Cutset$, $n_i^{1\to2}$ means all nodes are originally in V_1 and then its fan-out node is moved to V_2.
- $n_i^{2\to1}$: event that net n_i is removed from $Cutset$ by moving nodes to V_1. For a C-type net n_i in $Cutset$, $n_i^{2\to1}$ means all its nodes in V_2 are moved to V_1; for an FF-type net n_i in $Cutset$, $n_i^{2\to1}$ means its fan-out node is originally in V_2 and then all its fan-in nodes are moved to V_1.
- $p(n_i^{a\to b}|u)$: probability of removing net n_i from $Cutset$ by moving nodes to V_b in the condition that node u is originally in V_a and then is moved to V_b.
- $p(n_i^{a\to b}|u^c)$: probability of removing net n_i from $Cutset$ by moving nodes to V_b in the condition that node u is originally in V_b and then stays in V_b.
- f_{n_i} : fan-out node of net n_i.
- $S_a(u)$: set of successors of u in stage V_a.
- $A_a(u)$: set of ancestors of u in stage V_a.
- $E_a(n_i,n_j)$: set of nodes in V_a that are both in nets n_i and n_j. E.g., in Figure 9, $E_1(n_5,n_6)=\emptyset$ and $E_2(n_5,n_6)=\{v_5\}$.
- $N_a(u)$: set of u's neighbors in V_a.

- $I(u)$: set of nets which contain node u.
- $M_a(n)$: set of nets that have common nodes with net n in V_a. E.g., in Figure 9, $M_1(n_5) = \{n_4, n_7\}$ and $M_2(n_5) = \{n_2, n_6\}$.
- $e(n_i^{a\to b})$: expected gain in the condition that net n_i is moved from V_a to V_b.
- $e_{n_j}(n_i^{a\to b})$: expected gain contributed by n_j in the condition that net n_i is moved from V_a to V_b.
- $g(u)$: gain of node u.
- $g_{n_j}(u)$: gain of node u contributed by n_j.

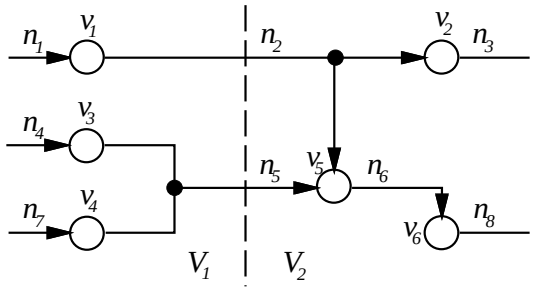


Figure 9: Example for the notation.

According to the definitions of $n_i^{1\to 2}$ and $n_i^{2\to 1}$, we have the following equations. For a C-type net n_i with node u, $u \in V_a$,

$$\begin{aligned} p(n_i^{a\to b}|u) &= \prod_{v\in N_a(n_i)-\{u\}} p(v) \\ p(n_i^{b\to a}|u^c) &= \prod_{v\in N_b(n_i)} p(v). \end{aligned}$$

For an FF-type net n_i with node u, $\forall v, v \in V_1$,

$$\begin{aligned} p(n_i^{1\to 2}|u) &= \begin{cases} 1 & \text{if } u = f_{n_i} \\ 0 & \text{otherwise} \end{cases} \\ p(n_i^{1\to 2}|u^c) &= \begin{cases} p(f_{n_i}) & \text{if } u \neq f_{n_i} \\ 0 & \text{otherwise} \end{cases} \\ p(n_i^{2\to 1}|u) &= 0. \end{aligned}$$

For an FF-type net n_i with node u, $f_{n_i} \in V_2$,

$$\begin{aligned} p(n_i^{2\to 1}|u) &= \begin{cases} \prod_{v\in N_2(n_i)-\{f_{n_i},u\}} p(v) & \text{if } u \in N_2(n_i) - \{f_{n_i}\} \\ 0 & \text{otherwise} \end{cases} \\ p(n_i^{2\to 1}|u^c) &= \begin{cases} \prod_{v\in N_2(n_i)-\{f_{n_i}\}} p(v) & \text{if } u \in N_1(n_i) \cup \{f_{n_i}\} \\ 0 & \text{otherwise} \end{cases} \\ p(n_i^{1\to 2}|u) &= 0. \end{aligned}$$

Moving a net n_i to some stage will affect the move of the other nets that have common nodes with net n_i. It is called the 2nd-order information [6]. Therefore, the expected gain for removing a net from $Cutset$ should be considered.

$$e(n_i^{a\to b}) = \sum_{n_j \in M_a(n_i)} e_{n_j}(n_i^{a\to b}).$$

For two C-type nets n_i and n_j, $n_i \cap n_j \neq \emptyset$,

$$e_{n_j}(n_i^{a\to b}) = c(n_j)p(n_j^{a\to b}) / \prod_{v\in E_a(n_i,n_j)} p(v). \tag{7}$$

For a C-type net n_i and an FF-type net n_j, $n_i \cap n_j \neq \emptyset$,

(1) if $f_{n_j} \in V_1$,

$$\begin{aligned} e_{n_j}(n_i^{1\to 2}) &= \begin{cases} c(n_j) & \text{if } E_1(n_i,n_j) = f_{n_j} \\ 0 & \text{otherwise} \end{cases} \\ e_{n_j}(n_i^{2\to 1}) &= 0. \end{aligned}$$

(2) if $f_{n_j} \in V_2$,

$$\begin{aligned} e_{n_j}(n_i^{2\to 1}) &= \begin{cases} \frac{c(n_j)p(n_j^{2\to 1})}{\prod_{v\in E_2(n_i,n_j)} p(v)} & \text{if } f_{n_i} \notin E_2(n_i,n_j) \\ 0 & \text{otherwise} \end{cases} \\ e_{n_j}(n_i^{2\to 1}) &= 0. \end{aligned}$$

For an FF-type net n_i and a C-type net n_j, $n_i \cap n_j \neq \emptyset$,

$$\begin{aligned} e_{n_j}(n_i^{1\to 2}) &= \begin{cases} c(n_j)p(n_j^{1\to 2})/p(f_{n_i}) & \text{if } f_{n_i} \in n_j \\ 0 & \text{otherwise} \end{cases} \\ e_{n_j}(n_i^{2\to 1}) &= \begin{cases} \frac{c(n_j)p(n_j^{2\to 1})}{\prod_{v\in E_2(n_i,n_j)} p(v)} & \text{if } f_{n_i} \notin n_j \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

For two FF-type nets n_i and n_j, $n_i \cap n_j \neq \emptyset$,

(1) if $f_{n_j} \in V_1$,

$$\begin{aligned} e_{n_j}(n_i^{1\to 2}) &= \begin{cases} c(n_j) & \text{if } f_{n_i} = f_{n_j} \\ 0 & \text{otherwise} \end{cases} \\ e_{n_j}(n_i^{2\to 1}) &= 0. \end{aligned}$$

(2) if $f_{n_j} \in V_2$,

$$\begin{aligned} e_{n_j}(n_i^{2\to 1}) &= \begin{cases} \frac{c(n_j)p(n_j^{2\to 1})}{\prod_{v\in E_2(n_i,n_j)-\{f_{n_i}\}} p(v)} & \text{if } f_{n_j} \notin n_i \\ 0 & \text{otherwise} \end{cases} \\ e_{n_j}(n_i^{1\to 2}) &= 0. \end{aligned}$$

If net n_j in the above cases is not in $Cutset$ originally and moved into $Cutset$ in condition of $n_i^{a\to b}$, the term $-c(n_j)$ should be incorporated into $e_{n_j}(n_i^{a\to b})$. For example, two C-type nets n_i and n_j, $n_j \in V_a$,

$$e_{n_j}(n_i^{a\to b}) = -c(n_j) + c(n_j)p(n_j^{a\to b}) / \prod_{v\in E_a(n_i,n_j)} p(v). \tag{8}$$

Using the above equations, we can compute $g_{n_i}(u)$ as follows:

(1) if n_i is C-type,

$$\begin{aligned} g_{n_i}(u) &= (c(n_i) + e(n_i^{a\to b}))p(n_i^{a\to b}|u) - \\ &\quad (c(n_i) + e(n_i^{b\to a}))p(n_i^{b\to a}|u^c). \end{aligned} \tag{9}$$

(2) if n_i is FF-type,

$$\begin{aligned} g_{n_i}(u) &= (c(n_i) + e(n_i^{a\to b}))p(n_i^{a\to b}|u) - \\ &\quad (c(n_i) + e(n_i^{a\to b}))p(n_i^{a\to b}|u^c). \end{aligned} \tag{10}$$

Thus, the gain of a node u is given by

$$g(u) = \sum_{n_i\in I(u)} g_{n_i}(u). \tag{11}$$

The probability of a node represents the likehood that the node will be moved. The node with a greater gain has a higher probability to be moved. Thus, we can get the probability of a node by a monotonically increasing mapping function of its gain. (In our experiments shown in the next section, we used an increasing linear function.) It causes an interdependency between probabilities and gains since we obtain the gains from probabilities of nodes as shown in the above equations. To break this endless recursive relation, we give each node the probability 0.5 in our experiment. Repeat computing gains and probabilities from each other until they are stable enough, and then we have initial probabilities (line 6 in Figure 10). In practice, three iterations are enough to reach a stable state. The probability-based algorithm PAT is summarized in Figure 10.

3.3 The Timing Constraint

The speed of a TMFPGA is determined by the maximum execution time of a micro-cycle. Therefore, we must reduce the longest path in a micro-cycle. In PAT, the lengths of the longest paths in both stages are upper bounded by $\lceil \frac{D}{2} \rceil$, where D is the length of the longest path in the circuit.

For a node v, let $\delta_O(v)$ denote the length of the longest path from v to primary outputs and $\delta_I(v)$ denote the length of the longest path from primary inputs to v. A node v cannot be put in V_1 if $\delta_I(v)$ is more than $\lceil \frac{D}{2} \rceil$, because there will exist a path of length more than $\lceil \frac{D}{2} \rceil$ from a primary input to v in V_1. For the same reason, a node v cannot be put in V_2 if $\delta_O(v)$ is more than $\lceil \frac{D}{2} \rceil$. According to the above rules, the nodes that may violate the timing constraint are fixed in proper stages before the clustering phase.

```
Algorithm: Prob(V, N)
Input: V—set of nodes; N—set of nets
Output: V1,V2—set of nodes;
\\ initial_partition(V)—a pair of node sets which satisfy
                        the precedence constraint
\\ Cutsize(V1, V2, N)—total weight of nets in Cutset
1   (V1, V2) ← initial_partition(V);
2   old_cutsize ← ∞;
3   min_cutsize ← Cutsize(V1, V2, N);
4   while (min_cutsize<old_cutsize);
5       old_cutsize ← min_cutsize;
6       P ← Comp_initial_prob(V1, V2, N);
7       G ← Comp_gain(V1, V2, N, P);
8       M ← Movable_nodes(V1, V2, N);
9       while (M ≠ ∅) do
10          u ← largest_gain(M);
11          if (u is feasible to be moved)
12              temporarily move u;
13              update G;
14              block_nodes(u);
15          new_cutsize ← Cutsize(V1, V2, N);
16          if (new_cutsize<min_cutsize)
17              min_cutsize ← new_cutsize;
18      Actually move the nodes that cause the minimum cutsize.
```

Figure 10: The 2nd phase of CPAT: PAT.

4 Experimental Results

The probability-based algorithm, PAT, and the clustering- and probability-based algorithm, CPAT, were implemented in the C++ language on a PC with a Pentium II 300 microprocessor and 128 MB RAM and tested on the MCNC Partitioning93 benchmark circuits. In Table 1, we compared PAT with the network-flow-based approach FBP-m [9] and the list scheduling List [2, 3] on the Xilinx TMFPGA model, in which a circuit was partitioned into eight stages. The size of a stage is bounded by the balance factor 5% (the same as in [9]). Columns 2 and 3 in Table 1 list the numbers of nodes and nets, respectively, in each circuit. Columns 4, 5, and 6 list the maximum numbers of micro registers used by List, FBP-m, and PAT, respectively. Columns 7 and 8 list the percentages of improvements of PAT over List and FBP-m, respectively. The results show that our PAT algorithm outperforms List and FBP-m by respective average reductions of 33.2% and 12.4% in the maximum numbers of micro registers required. It implies that the probability-based scheme is effective in reducing the interconnection for TMFPGAs.

In Table 2, we compare PAT and CPAT. Columns 2, 3, and 4 in Table 2 compare the maximum numbers of micro registers. Columns 5, 6, and 7 compare the runtimes. The results show that PAT has performance for smaller circuits while CPAT obtain better results for larger circuits. It implies that the clustering algorithm in CPAT leads to a considerable improvement as the size of a circuit increases over a certain bound. In addtion, the clustering algorithm in CPAT substantially reduces the problem size and thus the runtime. However, the clustering algorithm in CPAT might break the connectivities of nodes and nets when the circuit is small, in which the following probability-based algorithm might not be able to get the sufficient information to find a better result.

Circuit	#Nodes	#Nets	Max # of registers			PAT Imprv. (%)	
			List	FBP-m	PAT	List	FBP-m
c3540	1038	1016	177	166	126	+28.8	+24.0
c5315	1778	1655	265	165	157	+40.7	+5.1
c6288	2856	2824	117	114	114	+2.6	0
c7552	2247	2140	453	392	260	+42.6	+33.7
s820	340	314	91	81	43	+52.7	+46.9
s838	495	459	131	71	72	+64.8	-1.4
s1423	831	750	130	120	106	+18.5	+11.7
s9234	6098	5846	640	502	430	+32.8	+14.3
s13207	9445	8653	1118	901	838	+25.0	+7.0
s15850	11071	10385	1070	877	808	+24.5	+8.5
s35932	19880	17830	3806	2950	2138	+43.8	+27.5
s38417	25589	23845	3546	2892	2628	+25.9	+9.1
s38584	22451	20719	5131	2796	3611	+29.6	-25.7
Average						+33.2	+12.4

Table 1: Results for the 8-stages TMFPGA partitioning.

Circuit	Max # of registers			runtime (sec)		
	PAT	CPAT	Imprv. (%)	PAT	CPAT	Imprv. (%)
c3540	126	152	-17.1	3	3	0
c5315	157	174	-9.8	11	4	+63.7
s820	43	61	-29.5	4	2	+50.0
s838	72	93	-22.6	1	1	0
s1423	106	120	-11.7	3	2	+33.3
s9234	430	402	+6.5	29	25	+13.8
s13207	838	838	0	190	136	+28.4
s15850	808	767	+5.0	163	104	+36.2
s35932	2138	2018	+5.6	20131	15715	+21.9
s38417	2628	2468	+6.0	1125	926	+17.7
s38584	3611	1451	+59.8	1766	932	+47.2
Average			-0.7			+28.4

Table 2: Results for the 8-stages TMFPGA partitioning.

5 Conclusion

We have presented a clustering- and probability-based algorithm for the k-stage TMFPGA partitioning problem. Experimental results show that our probability-based algorithm outperforms the previous works, the List scheduling and the network-flow-based method, in a significant margin. Furthermore, we can improve the result and runtime by incorporating the clustering algorithm for large circuits.

Acknowledgments

The authors would like to thank Dr. Huiqun Liu for providing the benchmark circuits and helpful discussions on [9] and Prof. Ting-Chi Wang for his constructive comments.

References

[1] N.B. Bhat, et al., "Performance-oriented fully routable dynamic architecture for a field programmable logic dervice," Memmorandum No. UCB/RELM93/42, UC Berkeley, 1993.

[2] D. Chang and M. Marek-Sadowska, "Buffer minimization and Time-multiplexed I/O on Dynamically Reconfigurable FPGAs," *Proc. FPGA Symposium*, 1997, pp. 142–148.

[3] D. Chang and M. Marek-Sadowska, "Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs," *Proc. FPGA Symposium*, 1998, pp. 161–147.

[4] J. Cong, et al., "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering," *Proc. ICCAD*, 1997, pp. 441–446.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990, pp. 951–983.

[6] S. Dutt and W. Deng, “Partitioning Using Second-Order Information and Stochastic-Gain Functions,” *Proc. ISPD*, 1998, pp. 112–117.

[7] C.M. Fidducia and R.M. Mattheyses, ”A linear-time heuristic for improving network partitions”, *Proc. DAC*, 1982, pp. p175–181.

[8] B.W. Kernighan and S. Lin, ”An efficient heuristic procedure for partitioning graphs”, *Bell System Tech. Journal*, vol. 49, 1970, pp. 291–307.

[9] H. Liu and D. F. Wong, “Network Flow Based Circuit Partitioning for Time-multiplexed FPGAs,” *Proc. ICCAD*, 1998, 497–504.

[10] S. Trimberger, “A Time-Multiplexed FPGA,” *Proc. FCCM*, pp. 22–28, 1997.