Temporal Floorplanning Using the T-tree Formulation *†

Ping-Hung Yuh[1], *Chia-Lin Yang*[1], *Yao-Wen Chang*[2]

[1]Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
{r91089, yangc}@csie.ntu.edu.tw
[2]Dept. of Electrical Engineering & Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan
ywchang@cc.ee.ntu.edu.tw

Abstract

Improving logic capacity by time-sharing, dynamically reconfigurable FPGAs are employed to handle designs of high complexity and functionality. In this paper, we model each task as a 3D-box and deal with the temporal floorplanning/placement problem for dynamically reconfigurable FPGA architectures. We present a tree-based data structure, called *T-trees*, to represent the spatial and temporal relations among tasks. Each node in a T-tree has at most three children which represent the dimensional relationship among tasks. For the T-tree, we develop an efficient packing method and derive the condition to ensure the satisfaction of precedence constraints which model the temporal ordering among tasks induced by the execution of dynamically reconfigurable FPGAs. Experimental results show that our tree-based formulation can achieve significantly better solution quality with less execution time than the most recent state-of-the-art work.

1 Introduction

A Field Programmable Gate Array (FPGA) typically consists of regular identical reconfigurable cells (logic blocks) and interconnects around these blocks. Traditionally, an FPGA can only implement circuits by loading the serial configuration bit-streams into the chip at the starting time, and the reconfiguration must be done in a whole. Recently, various new architectures have been proposed by various vendors, such as the Atmel AT40K series [4], the Xilinx XC6200 series [10] and the Xilinx Virtex II series [18]. These new-generation FPGAs are partitionable and partially reconfigurable, allowing several tasks and circuits to share the same physical locations at different times and part of the chip to be reconfigured at run-time.

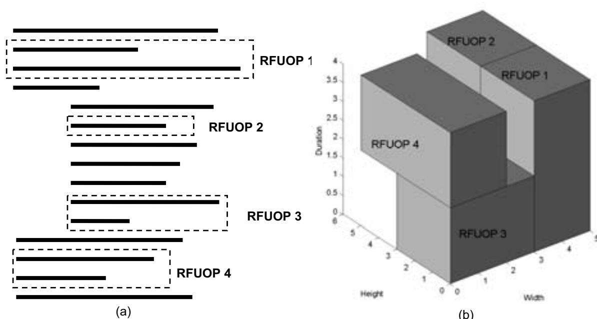

Figure 1: (a) A running program. (b) A 3D-placement of the running program.

Due to the capability of partially reconfigurable of recent FPGAs, studies have shown that an FPGA-based computation hardware system can improve performance for many applications [9, 16]. A reconfigurable system usually consists a host processor and an FPGA coprocessor called *reconfigurable function unit (RFU)* [5]. During the execution of a program, an RFU may have several configurations at different times. Figure 1(a) shows a program code that can be mapped into four RFU operations (*RFUOPs* or *modules*). Each line represents one line of the program code. Since the RFUOP must be placed on the RFU and has its own execution time, we may denote each RFUOP as a 3D-box, with its width and height (X and Y dimensions) representing the physical dimensions occupied by the RFUOP and its duration (Z dimension) being the execution time required for the operation. Because of the area constraint, we may not load all RFUOPs at the same time. Thus, at time 2 of the example shown in Figure 1, RFUOP 3 is swapped out and RFUOP 4 is swapped in. The question of how to place these RFUOPs becomes a 3D-placement problem. Each module is represented as a 3D-box with the spatial dimensions X and Y and the temporal dimension T. There exists temporal ordering constraints among tasks because one task's input may be another task's output. The goal of temporal floorplanning is to schedule all modules on an RFU so that the specified objective function (e.g., the product of chip area and execution time—the volume of the 3D floorplan/placement) is optimized and no two modules violate the temporal constraints.

One significant purpose of a temporal floorplanner is to be a scheduler. For some applications, the flow of the program has already been known in advance (for example, in DSP applications). Thus, the scheduler can schedule all RFUOPs that must be executed on the RFU before the program starts. Also, the scheduler can perform various optimizations on the configuration of the RFU, such as the minimization of the reconfiguration overhead.

1.1 Previous Work

Teich et al. [15] first introduced a *component graph* to solve the 3D placement problem, assuming that there exists no temporal ordering among modules. However, in the real world, there exist temporal relations among modules. Thus, Fekete et al. [7] extended their idea and solved the 3D-placement problem with temporal precedence constraints by using the dependency graph. Bazargan et al. in their pioneering work [5] considered both the offline placement (3D template placement) and the online placement problems. In the offline placement, they modelled each RFUOP as a 3D box and fixed the width and height of the RFU. They proposed a 3D floorplanner which implements four effective methods, including one greedy method called KAMER-BF (Keep All Maximal Empty Rectangles with Best Fit). In the online placement, they allocated the free space of an RFU to an RFUOP dynamically based on different greedy methods (e.g., the best-fit and first-fit heuristics). The work [1] divided an RFU into several horizontal strips. This formulation simplifies the traditional 2D placement problem as a 1D (linear) placement problem, facilitating a faster placement and routing engine. Recently, Yuh et al. [19] first proposed a graph-based topological representation, called *3D-subTCG*, to handle the temporal floorplanning problem. The 3D-subTCG uses three transitive closure graphs (one for each dimension) to represent a 3D placement.

1.2 Our Contributions

In this paper, we propose the first tree-based formulation, called *T-trees*, to model both the temporal and spatial relations among tasks to solve the 3D-floorplanning/placement problem. Given a compacted placement that cannot move toward the origin, we can construct its corresponding ternary T-tree in linear time. In comparison with 3D-subTCG [19] which is the fastest representation for the 3D floorplanning/placement problem in the literature, T-trees have the following advantages:

- Since the operations on a 3D-subTCG are performed on edges,

*Chia-Lin Yang's work was partially supported by the National Science Council of Taiwan under Grant No's. NSC 93-2220-E-002-001 and NSC 93-2752-E-002-008-PAE.

†Yao-Wen Chang's work was partially supported by the National Science Council of Taiwan under Grant No's. NSC 92-2215-E-002-018, NSC-93-2220-E-002-001, and NSC 93-2752-E-002-008-PAE.

its time complexity is $O(n^2)$, where n is the number of nodes/modules. In contrast, the operations on a T-tree are performed on nodes; therefore, the time complexity is only $O(n)$.

- Based on the T-tree representation, we can derive a more efficient packing method than that used by 3D-subTCG. T-trees show about 15X speedup over 3D-subTCG for packing 3D-ami49 (the largest circuit used in [19]).
- There are only $O(n!\frac{3^{3n}}{2^{2n}n^{1.5}})$ combinations of a T-tree. Although the authors of 3D-subTCG [19] do not derive its solution space, we observe that a 3D-subTCG has $O((n!)^3)$ combinations.

To handle the precedence constraints among tasks, we derive an effective and efficient method to examine the feasibility of a T-tree. The structure of T-tree presents a nice property that enables easy feasibility detection; for example, if node n_j is in the left subtree of node n_i, task v_j must be executed after task v_i. Based on this property, we can perform feasibility detection in $O(h)$ time, where h is the height of a T-tree. (Note that we treat the number of precedence constraints as a constant.) If a T-tree results in an infeasible placement, the T-tree is re-constructed to remove the violated conditions. To reduce the probability that a T-tree results in an infeasible placement after an operation, we filter out a set of operations that will definitely introduce precedence violations. We also derive in this paper the solution space of T-tree and prove the reachability of the solution space. The study provides a solid theoretical foundation for the effectiveness and efficiency of the simulated annealing (SA) based optimization process used in our temporal floorplanner. Experimental results show that our T-tree based SA scheme consistently obtains much better results in shorter running time than the 3D-subTCG approach. For a large circuit of 300 tasks and 120 precedence constraints, for example, the T-tree based SA scheme obtains a solution of 13.7% deadspace in less than 1.15 hours, while the 3D-subTCG method needs about 7.51 hours and results in a solution of 34.2% deadspace.

In addition to the classical 3D-floorplanning problem that minimizes the product of the area and execution time (i.e., the volume of the 3D floorplan/placement), we also propose in this paper a novel T-tree based SA mechanism to handle the fixed-outline floorplanning problem, for which the area of a reconfigurable device is fixed.

The fixed-outline floorplanning problem was advocated by Kahng in [12] to address modern floorplanning constraints. Adya and Markov in [2] and [3] first proposed algorithms for the classical 2D fixed-outline floorplanning problem. They added penalty to the cost function for the modules that are placed out of the desired outline. In this paper, we extend the idea to handle the fixed-outline temporal (3D) floorplanning problem. For this problem, we propose a new objective function to guide simulated annealing. Moreover, we bias the selection of operations performed in each SA iteration to increase the probability of success of satisfying the fixed outline constraint. Experimental results show that our fixed-outline temporal floorplanner significantly improves the success rate of fitting 3D-boxes into the fixed outline.

The remainder of this paper is organized as follows. Section 2 formulates the temporal floorplanning problem. Section 3 introduces the T-tree representation. Section 4 describes our temporal floorplanning algorithm. In Section 5, we derive the solution space of T-tree and prove the reachability of the solution space. Section 6 details our fixed-outline floorplanning method. Section 7 reports the experimental results. Finally, conclusions are given in Section 8.

2 Formulation

In the reconfigurable architecture, a *task* v is loaded into the device for a period of time for execution. Therefore, each task can be represented as a 3D module with spatial dimension X and Y and the temporal dimension T. Throughout this paper, we use task and module interchangeably. Let $V = \{v_1, v_2, ..., v_m\}$ be a set of m tasks whose widths, heights, and execution times are given by W_i, H_i, and T_i, $1 \le i \le m$. We use (x_i, y_i) $((x'_i, y'_i))$ to denote the coordinate of the bottom-left (top-right) corner of a task v_i, and t_i (t'_i) the starting (ending) time of task v_i, $1 \le i \le m$, scheduled in the reconfigurable device. These tasks often need to be executed in a specific order because one task's input could be another task's output. The temporal ordering among tasks is referred to as the *precedence constraint* in the 3D floorplanning problem. Let $D = \{(v_i, v_j) | 1 \le i, j \le m, i \ne j\}$ denote the precedence constraint for the tasks v_i and v_j (i.e., v_i must be executed before v_j). The precedence constraints should not be violated during floorplanning/placement.

In order to measure the quality of a floorplan, we consider the same objectives as those of [7] and [19], i.e., volume, wirelength, communication and reconfiguration overheads. The definitions of these four objective functions are given below:

- **Volume (the minimum bounding box of a placement):** In temporal floorplanning, we need to consider the trade-off between the area of a device and the total execution time. If we use a larger device, the total execution time could be shortened. In contrast, it takes longer if a smaller device is used. Therefore, we shall minimize the product of the area of the device and the total execution time, i.e., the volume of a 3D floorplan/placement.
- **Wirelength (the summation of half bounding box of interconnections):** Due to the special architecture of the reconfigurable device, the method to estimate the wirelength in the temporal floorplanning is different from the traditional floorplanning/placement problem. Given a net, those nodes in the net may be executed at the same time or at different times. If they are executed at the same time, we can estimate the wirelength according to their geometric distance directly. However, we have to project all nodes onto the same time frame before computing their wirelength if they are executed at different time frames.
- **Communication overhead:** We quantify the communication overhead based on the Xilinx Virtex XCV1000-like architecture. Similar to the work by Fekete et al. [7], we assume that a task communicates with another task (data-dependence) in the following way: the results of a CLB, which are read by the succeeding task, are first written to external memory through a bus interface. The dependent task, which has been loaded at the specified position, then perform a read-in of the results. Recall that a *frame* is the atomic unit that can be written to or read from. Each frame contains 1248 bits and the bus width is of only 8 bit. Thus, it takes approximately $1248/8 + 24 = 180$ clock cycles in each read-in or read-out, where the 24 cycles are used to configure the bus interface as described on the Xilinx FPGA data book [17]. Therefore, the communication overhead of each reconfiguration takes $360 \times f$ clock cycles (we should first write the data to the external memory and then read back the data) if data in f columns need to be transferred.
- **Reconfiguration overhead:** As described in [17], the Xilinx Virtex-series FPGA is column-oriented (i.e., all bits in one column should be updated in each read-in or read-out). Suppose that a task v_i occupies $W_i \times H_i$ CLBs. We have to reconfigure H_i columns of CLBs in each reconfiguration. As an example, each CLB column in a Virtex FPGA consists of 48 frames, which takes $(1248/8) \times 48 + 24 = 7512$ clock cycles to configure one CLB column. This means that we need $7512 \times W_i$ clock cycles in total if the addresses in the column are incrementally updated.

In this paper, we treat a task v_i as a 3D box. A placement $\mathcal{P}$ is an assignment of (x_i, y_i, t_i) for each v_i, $1 \le i \le m$, such that no two boxes overlap and all precedence constraints are satisfied. The goal of temporal floorplanning is to optimize a predefined cost metric (defined in the above) induced by a placement.

3 The T-tree Representation

Chang et al. [6] first proposed a binary tree-based 2D floorplanning representation, called *B*-trees*. Each node of the B*-tree has at most two children that represent the dimensional relationship among modules. T-trees are inspired by B*-trees, allowing each node with at most three children that represent the dimensional relationship among modules, as shown in Figure 2. The T-tree represents the geometric relationships between two modules as follows. If node n_j is the left child of node n_i, module v_j must be placed adjacent to module v_i on the T^+ direction, i.e., $t_j = t_i + T_i$. If node n_k is the middle child of node n_i, module v_k must be placed in the Y^+ direction of module v_i, with the t-coordinate of v_k equal to that of v_i, i.e., $t_k = t_i$ and $y_k > y_i$. If node n_l is the right child of node n_i, module v_l must be placed on the X^+ direction of

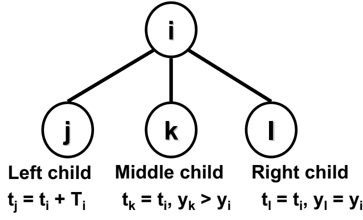

Figure 2: The structure of a T-tree.

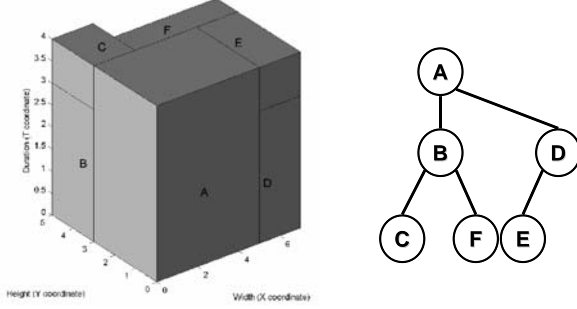

Figure 3: A compacted placement and the corresponding T-tree

module v_i, with the t- and y-coordinates equal to those of v_i, i.e., $t_l = t_i$ and $y_l = y_i$.

Below we describe how to transform between a placement and its corresponding T-tree.

3.1 From a Compacted Placement to its T-tree

Given a compacted placement, we can represent it by a unique T-tree. A placement is said to be *compacted* if and only if no module can be moved along its X^-, Y^- or T^- directions while other modules are fixed. The root of the tree corresponds to the task placed at the origin. We construct a T-tree for a compacted placement in a DFS manner: Starting from the root, we recursively construct the left subtree, then the middle subtree, and finally the right subtree. Let R_i denote the set of tasks that are adjacent to v_i in the T^+ direction . The left child of node n_i corresponds to the lowest task of R_i in the X-Y plane. The middle child of node n_i corresponds to the first task in the Y^+ direction, with its t-coordinate equal to that of n_i. The right child of node n_i represents the first task in the X^+ direction, with its y- and t-coordinates equal to those of n_i. A compacted placement can be transformed to its corresponding T-tree in linear time.

We use the placement shown in Figure 3 to demonstrate how to construct the corresponding T-tree. We choose n_a as the root of the T-tree since task v_a is on the bottom-left corner of the placement. Since no task is adjacent to v_a in the T^+ direction, node n_a does not have any left child. We then build the middle subtree of n_a. The middle child of n_a is n_b because task v_b is adjacent to v_a in the Y^+ direction and $t_b = t_a$. The left child of n_b is n_c because $t_c = t_b + T_b$, and the right child of n_b is n_f because task v_f is adjacent to v_b in the X^+ direction with $t_f = t_b$ and $y_f = y_b$. Similarly, we can construct the right subtree of n_a.

The above tree-construction method leads to the following theorem.

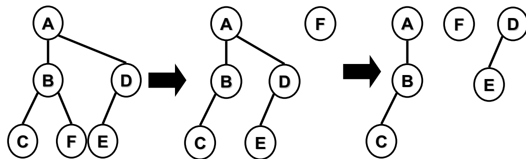

Figure 4: The T-tree decomposition process

Theorem 1 *There exists a unique correspondence between a compacted placement and its induced T-tree.*

3.2 From a T-tree to its Placement

Now we describe the packing method for a T-tree. The t-coordinate of each module can easily be obtained by traversing the T-tree in the DFS order. If node n_j is the left child of node n_i, $t_j = t_i + T_i$; otherwise, $t_j = t_i$. Once the t-coordinates are fixed, we can utilize the existing tree solutions in [8] and [6] to compute y coordinates. We first decompose a T-tree into a set of binary trees. The T-tree decomposition process is shown in Figure 4. Starting from the root, we traverse a T-tree in the DFS order. When we encounter a node which has the right child, n_b in the example shown in Figure 4, we decompose the tree into two subtrees: one is the right subtree of n_b, and the other is the original tree without the right subtree of n_b. The same decomposition procedure is applied to each subtree until a leaf node is encountered. For each binary tree, we adopt the *contour* data structure presented in [8] and [6] to determine the y-coordinate of each module. The contour structure is a doubly-linked list of modules that records the contour line in current compaction. To compute x coordinates, we maintain a list L to store all tasks whose t- and y-coordinates are already determined. The x-coordinate of task v_i is equal to $max\{x'_k|$ the projections of v_k and v_i are overlapped on the Y-T plane for $k \in L\}$.

The time complexity of the T-tree packing method is $O(n^2)$, which is bounded by the computation of x coordinates. Although the complexity is the same as the 3D-subTCG [19], in practice, we observe about 15x speedup over 3D-subTCG when packing 3D-ami49 (the largest circuit used in [19]).

The 3D-subTCG representation uses three transitive closure graphs (one for each dimension) to represent a placement. Given n nodes, a 3D-subTCG has $\frac{n(n-1)}{2}$ edges. Therefore, each transitive closure graph has $\frac{n(n-1)}{6}$ edges on average. The time complexity for computing one coordinate for each task is $O(n^2)$ by applying a well-known *longest path algorithm* [14]. For the T-tree representation, the computation of t and y coordinates takes only $O(n)$ time. Therefore, based on the T-tree, we are able to develop a more efficient packing method than the 3D-subTCG. Consequently, we have the following theorem and property.

Theorem 2 *Given a T-tree, there exists a feasible 3D placement corresponding to the T-tree.*

Property 1 *If node n_j is in the left subtree of node n_i, task v_j must be executed after task v_i.*

4 Temporal Floorplanning Algorithm

Our floorplanning algorithm is based on the simulated annealing method [13]. The cost function Φ used in the algorithm is given by

$$\Phi = \alpha V + \beta W + \gamma O, \tag{1}$$

where V stands for the volume of the placement, W is the total wirelength, O is the reconfiguration and communication overheads, and α, β, γ are user-specified constants. Given a T-tree (a feasible solution), we perturb the T-tree to obtain another feasible T-tree by using the following three operations:

- *Move:* move a task to another place.
- *Swap:* swap two tasks.
- *Rotate:* rotate a task.

Since the resulting T-tree after perturbation may violate the precedence constraints, we need to perform feasibility checking on the resulting T-tree and re-construct the tree if any of the temporal constraints is violated. Below we discuss the details of the three operations and feasibility detection.

4.1 Move

The *Move* operation needs the *Insertion* and *Deletion* operations for inserting and deleting a node to and from a T-tree.

Deletion

The target node for deletion could be as follows:

- Case 1: a leaf node,

- Case 2: a node with only one child, or
- Case 3: a node with two or three children.

For the first case, the target node is simply deleted. For Case 2, we delete the target node and place its only child at the position of the deleted node. This operation can be done in $O(1)$ time. For Case 3, the target node n_i is deleted, and one of its children n_c is moved to the original position of n_i. Then we move one child of n_c to the original position of n_c. This process proceeds until a leaf node is encountered. This operation takes $O(h)$ time, where h is height of the T-tree.

Insertion

When adding a task, we may place it around some tasks. There are two types of positions that can be inserted, the *internal position* and the *external position*. The internal position is a position between two nodes in a T-tree while the external one is a position that is pointed by a NULL pointer. A node can be inserted into one of these two positions.

4.2 Swap

For swapping two nodes in a T-tree, we simply exchange their parent and child nodes.

4.3 Rotation

A module can only be rotated on the X-Y plane because the execution time of a task is fixed. To perform the rotation operation, we simply exchange the width and height of a node.

4.4 Feasibility Detection and Tree Re-construction

To maintain the temporal ordering among tasks, we need to guarantee that a T-tree meets all the precedence constraints after each perturbation. For the three operations mentioned above, Move and Swap might violate the temporal constraints. Therefore, in this section, we describe how to examine the feasibility of a T-tree and propose a procedure to re-construct a T-tree to meet the precedence constraints.

From Property 1, we know that if node n_j is in the left subtree of n_i, task v_j must be executed after task v_i. Therefore, to ensure all the precedence constraints are not violated, a node n_k must be placed in the left subtree of n_p, where n_p has the latest ending time among the tasks that must be executed before task v_k. Therefore, the feasibility detection can be summarized in the following theorem:

Theorem 3 *Let $I_k, 1 \leq k \leq n$, denote the set of tasks that must be executed before task v_k. If node n_k is in node n_p's left subtree, where $t'_p = max\{t'_i | v_i \in I_k\}$, then v_k is guaranteed to satisfy the precedence constraint.*

Once we identify a node that violates the precedence constraint, we re-construct the T-tree to remove the violation conditions. Assume task v_i violates precedence constraints and v_p is the task that has the latest ending time in I_i. Let $U = \{$all nodes in the left subtree of $n_p\} \cup \{n_p\}$. In U, we look for a node n_j that minimizes $|t_j - t_i|$ with $I_j = \emptyset$. If $n_j \neq n_p$, n_i is swapped with n_j; otherwise, it means that $n_j = n_p$ or $I_j \neq \emptyset$ for every $n_j \in U$. In this case, we make n_i the left child of n_p. The tree re-construction process is summarized in Figure 5:

Algorithm: Tree Re-construction(H, D)
H: a T-tree;
D: the set of precedence constraints
1 **begin**
2 **repeat**
3 Scan all pairs of tasks in D
4 **if** task v_k violates precedence constraints **then**
5 Find n_p with $t'_p = max\{t'_i | v_i \in I_k\}$;
6 $U = \{$all nodes in the left subtree of $n_p\} \cup \{n_p\}$;
7 $n_j = min\{|t_j - t_k| | n_j \in U, I_j = \emptyset\}$;
8 **if** $n_j \neq n_p$ **then** swap nodes n_j and n_k
9 **else** make n_k the left child of n_p;
10 Perform packing for H
11 **until** all tasks satisfy the precedence constraint
12 **end**

Figure 5: Summary of the tree re-construction process.

To avoid infeasible T-trees that may trigger tree re-construction, we design a mechanism to filter out the operations that will definitely cause precedence violations. Consider the T-tree shown in Figure 6. Assume v_b must be executed before v_e. Node n_e can only be swapped with a node in n_b's left subtree, or be inserted into an internal/external position in n_b's left subtree. Further, node n_b cannot be swapped with any node in the subtree rooted by n_e or be inserted into any internal/external position in the subtree rooted by n_e. Based on this observation, we number each node in a T-tree in the depth-first search (DFS) order starting from the right subtree. Each node is associated with two values, DFS_up and DFS_low. Let O_k, $1 \leq k \leq n$, be the set of tasks that must be executed after v_k. A node n_k's DFS_low is the DFS number of n_p's left child, where $t'_p = max\{t'_i | v_i \in I_k\}$. Similarly, n_k's DFS_up is n_q's DFS number, where $t_q = \min\{t_i | v_i \in O_k\}$. For the Swap and Move operations performed on n_k, we heuristically choose nodes in the range of $[DFS_low, DFS_up)$ based on the above observation.

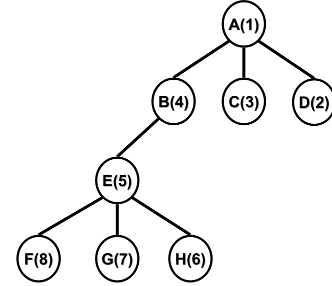

Figure 6: A T-tree and its DFS order.

5 Solution Space and Reachability

5.1 Solution Space

The total number of combinations of an n-node T-tree can be computed by the number of different unlabled n-node 3-ary tree and the permutation of n labels. The permutation of n labels is $n!$. From [11], the counting of unlabled p-ary tree with n nodes is given by

$$\frac{1}{(p-1)n+1}\binom{pn}{n}. \tag{2}$$

Applying Stirling's approximation, we have

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n. \tag{3}$$

Setting p equal to 3 in Equation (2), we can obtain the following asymptotic form:

$$O(\frac{3^{3n}}{2^{2n}n^{1.5}}). \tag{4}$$

Thus, the total number of possible placements for an n-node T-tree is

$$O(n!\frac{3^{3n}}{2^{2n}n^{1.5}}). \tag{5}$$

Based on the above discussion, we have the following theorem:

Theorem 4 *The size of the solution space of a T-tree is $O(n!\frac{3^{3n}}{2^{2n}n^{1.5}})$.*

5.2 Reachability

For a well-structured solution space, it should have the property that there exists a series of operations to transform between two arbitrary solutions. For such solution structure, it is possible to find an optimal solution from any initial solution in the solution space. Two placements are said to be *eqivalent* if the topologies of their corresponding T-trees, the labelling for each node, and the orientations of all the tasks, are the same. Below we prove the reachability of the solution space of the T-tree.

Theorem 5 *Given two T-trees H_1 and H_2, H_1 can be transformed to H_2 via at most 4n-3 operations.*

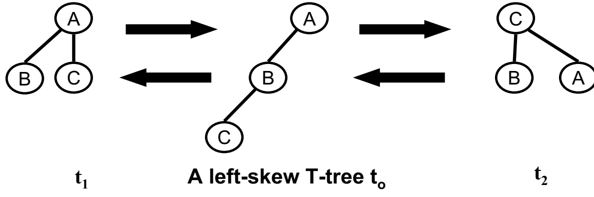

Figure 7: The transformation process.

Circuit	# of modules	# of pads	# of nets	# of pins
3D-apte	9	73	97	214
3D-xerox	10	107	203	696
3D-hp	11	43	83	264
3D-ami33	33	42	123	480
3D-ami49	49	24	408	931

Table 1: The 3D-MCNC benchmark.

Circuit	# of modules	# of pads	# of nets	# of pins	# of precedence constraints
3D-n100	100	334	885	1873	49
3D-n200	200	564	1585	3599	88
3D-n300	300	569	1893	4358	120

Table 3: The three 3D-GSRC benchmarks.

Proof:

To transform H_1 to H_2, we first transform H_1 to a tree with the same topology as H_2. We can then get the correct label for each node by the Swap operation and the correct orientation for each task by the Rotation operation.

If there exists a T-tree H_0 to which both H_1 and H_2 can be transformed via a series of operations, and the operations are reversible, then H_1 can be transformed to H_2 with H_0 as an intermediate solution. The transformation process is shown in Figure 7. The Swap and Rotation operations are always reversible, but the Move operation is reversible only if the target node is a leaf and the destination is an external position. Thus, we choose a *left-skew* T-tree as the intermediate solution. A left-skew T-tree is a T-tree in which each node has only a left child.

H_1 can be transformed to H_0 through a series of operations that recursively move the rightmost leaf of H_1 to its leftmost external position. It needs at most $n-1$ Move operations to transform H_1 to H_0 and another $n-1$ Move operations from H_0 to H_2. Then we need at most $n-1$ Swap operations to get the correct label for each node and at most n Rotation operations to get the correct orientation for each task. As a result, the total number of operations required to transform H_1 to H_2 is at most $4n-3$.

6 Fixed-outline Floorplanning

For fixed-outline floorplanning, the area of the reconfigurable device is fixed. Let W_f/H_f and W_p/H_p denote the width and height of a reconfigurable device and a placement, respectively. A feasible placement of fixed-outline floorplanning must satisfy the outline constraint; that is, $W_p \leq W_f$ and $H_p \leq H_f$. Therefore, we consider excessive volumes of a placement in the objective function for the fixed-outline floorplanning problem. The new objective function Φ' is given by

$$\Phi' = \alpha V + \beta W + \gamma O + \delta F, \qquad (6)$$

where δ is also a user-specified constant, and F is given by the following equation:

$$\begin{aligned} F \;=\; & min((\max(W_p - W_f, 0) \times H_p \times Time) + \\ & (\max(H_p - H_f, 0) \times W_p \times Time), \\ & (\max(W_p - W_f, 0) \times H_p \times Time) + \\ & (\max(H_p - H_f, 0) \times W_p \times Time)), \qquad (7) \end{aligned}$$

where $Time$ is the total execution time for a placement. Since the whole design can be rotated by 90 degrees, we choose the smaller excessive volume of two orthogonal placements.

Besides considering the excessive volume in the objective function, we bias the selection of the destination of the Move operations based on the value k/n, where k is the number of infeasible placements in the last n iterations. In our experiments, we set n equal to 500. A large k/n value indicates that the placement is not easy to fit into the device outline; therefore, we should try to place a module along the T direction to increase the success probability. In contrast, if the k/n value is small, we should try to place a module in the X or Y direction to minimize the task execution time.

7 Experimental Results

Based on simulated annealing [13], we implemented our T-tree based temporal floorplanning algorithm in the C++ programming language on a 1GHz SUN Blade 2000 machine with 1GB memory. We also implemented the 3D-subTCG based temporal floorplanner [19] with the same simulated annealing engine and on the same SUN Blade 2000 machine.

For the outline-free problem, we conducted two experiments. For the first experiment, we applied the five 3D-MCNC benchmark suites used in [19]. Table 1 lists the five 3D-MCNC benchmarks. To test the circuits with more constraints, we further constructed five additional benchmarks, 3D-apte-2, . . ., 3D-ami49-2. Table 2 shows the experimental result compared with the 3D-subTCG. Note that α, β and γ were all set to 1 for all circuits in order to be consistent with [19]. We can see that, compared with the 3D-subTCG, the T-tree can achieve smaller deadspace (10.4 % vs. 12.82% for the original set of MCNC benchmarks from [19] and 15.2% vs. 18.4% for the additional set of MCNC benchmarks) in shorter running time (18.17 sec vs. 58.5 sec and 18.49 sec vs. 77.05 sec). Figure 8 shows the resulting placement of 3D-ami49 with 11 precedence constraints.

To test the scalablility of the T-tree, in the second experiment, we used the GSRC benchmarks. The circuits in the GSRC benchmark are much larger than those in the 3D-MCNC benchmark. We added task execution times and precedence constraints by ourselves. The GSRC benchmark with execution times and precedence constraints are referred to as 3D-GSRC in this paper. The information of the 3D-GSRC circuits is summarized in Table 3. Table 4 shows the experimental result. We can see that for 3D-n300, the largest circuit in the GSRC with 300 tasks and 120 precedence constraints, the T-tree based SA scheme obtains a solution of 13.7% deadspace in less than 1.15 hours, while the 3D-subTCG method needs about 7.51 hours and results in a solution of 34.2% deadspace.

For the fixed-outline floorplanning problem, we chose the 3D-n100 circuit for experiments. We added various outline constraints. Table 5 reports the success rate [1] and the task execution time [2] of the fix-outline SA engine described in Section 6. We also list the results from the outline-free SA engine for comparison. In this experiment, we set different ratios of desired widths and heights for 3D-n100. It shows that the fixed-outline SA engine achieves much higher success rate compared with the outline-free engine. According to the design of the fixed-outline SA engine, we can achieve much higher success rates (88.75% vs. 47.37%) at the expense of 5% longer task execution time. The results show the effectiveness of the fixed-outline SA engine.

8 Conclusions

We have proposed the T-tree representation for handling the temporal floorplanning/placement problem. Compared with the 3D-subTCG, the T-tree can achieve better solution quality in less time. This makes it suitable for large circuits. The main reasons why the T-tree is more efficient than the 3D-subTCG is three-fold: (1) the packing time of T-tree is faster, (2) the size of the solution space is smaller than the 3D-subTCG, and (3) the solution space of the T-tree is well-structured. Thus, the annealing engine can devote more time to explore the solution space, resulting in better solution quality and less running time.

One of the abilities of recent FPGA architectures is that they can be reconfigured on the chip (internally), thus reducing the reconfiguration

[1] Number of runs that satisfies the fixed-outline constraint in 100 runs

[2] The minimum total execution time in all successful runs

Circuit	Total volume	# of constraints	3D-subTCG				T-tree			
			Volume	Wire length (mm)	Dead space (%)	Time (sec.)	Volume	Wire length (mm)	Dead space (%)	time (sec.)
3D-apte	9.88×10^7	3	1.05×10^8	252.5	5.9	0.91	1.05×10^8	252.5	5.9	0.35
3D-xerox	4.05×10^7	3	4.42×10^7	577.3	8.4	1.32	4.39×10^7	338.0	7.8	0.31
3D-hp	1.29×10^7	3	1.50×10^7	163.6	13.7	6.07	1.41×10^7	165.9	8.6	1.31
3D-ami33	2.32×10^6	7	2.82×10^6	59.7	17.8	115.51	2.77×10^6	56.6	16.2	28.72
3D-ami49	1.32×10^8	11	1.61×10^8	669.5	18.3	168.69	1.52×10^8	672.4	13.5	60.18
Average					12.82	58.5			10.4	18.17
3D-apte-2	9.88×10^7	4	1.05×10^8	252.5	5.9	0.96	1.05×10^8	252.5	5.9	0.37
3D-xerox-2	4.05×10^7	4	4.87×10^7	471.9	16.8	4.76	4.72×10^7	405.3	14.3	0.69
3D-hp-2	1.29×10^7	6	1.65×10^7	145.1	21.9	4.61	1.64×10^7	139.0	20.9	0.58
3D-ami33-2	2.32×10^6	15	3.16×10^6	55.4	26.5	60.69	2.92×10^6	49.4	20.7	28.13
3D-ami49-2	1.32×10^8	21	1.67×10^8	768.3	20.9	314.24	1.54×10^8	686.0	14.6	62.70
Average-2					18.4	77.05			15.2	18.49

Table 2: Results of volume and wirelength optimization for the five 3D-MCNC benchmarks (volume = mm^2 x clockcycles).

Circuit	Total volume	3D-subTCG				T-tree			
		Volume	Wire length (mm)	Dead space (%)	Time (sec.)	Volume	Wire length (mm)	Dead space (%)	time (sec.)
3D-n100	5.28×10^6	6.61×10^6	247.5	20.1	1086.89	6.07×10^6	158.0	11.4	388.61
3D-n200	5.27×10^6	6.22×10^6	540.6	15.3	8658.41	5.93×10^6	420.2	11.2	1873.85
3D-n300	8.19×10^6	1.24×10^7	1146.0	34.2	27064.23	9.49×10^6	822.5	13.7	4175.19
Average				23.2	12269.84			12.16	2145.88

Table 4: Results of volume and wirelength optimization for the three 3D-GSRC benchmarks (volume = mm^2 x clockcycles).

overhead dramatically. One of the future work lies in utilizing internal registers to reduce the reconfiguration overhead.

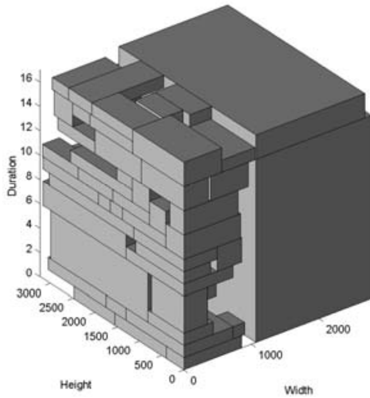

Figure 8: The result of 3D-ami49 with simultaneous volume and wirelength optimization.

Circuit name	Width	Height	Fixed-outline SA engine		Outline-free SA engine	
			Success rate	Exec. time (clk cycles)	Success rate	Exec. time (clk cycles)
3D-n100	200	145	97%	300	69%	270
	190	140	94%	270	39%	270
	165	160	93%	300	65%	270
	140	200	93%	270	40%	270
	140	180	90%	300	69%	270
	135	210	88%	270	21%	270
	160	160	85%	270	49%	270
	155	160	80%	300	27%	270
Average			88.8%	1.05	47.4%	1.0

Table 5: Results for various aspect ratios of desired widths and heights for 3D-n100 circuit.

References

[1] C. Ababeii and K. Bazargan, "Non-Contiguous Linear Placement for Reconfigurable Fabrics," *Proc. RAW*, Apr. 2004.

[2] S. N. Adya and I. L. Markov, "Fixed-Outline Floorplanning Through Better Local Search," *Proc. ICCD*, pp.328–334, 2001.

[3] S. N. Adya and I. L. Markov, "Fixed-outlin Floorplanning: Enabling Hierarchical Design," *IEEE Trans. on VLSI*, vol. 11, no. 6, pp. 1120–1135, Dec. 2003.

[4] Atmel, "AT40K05102040AL_Complete," Atmel, Inc.

[5] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design & Test of Computers*, vol. 17, no. 1, pp. 68–83, Mar. 2000.

[6] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-trees: A New Representation for Non-slicing Floorplan," *Proc. DAC*, pp. 458-462, June 2000.

[7] S. P. Fekete, E. Köhler, and J. Teich, "Optimal FPGA Module Placement with Temporal Precedence Constraints," *Proc. DATE*, pp. 658-665, Mar. 2001.

[8] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An O-tree Representation of Non-slicing Floorplan and Its Application," *Proc. DAC*, pp. 268-273, June 1999.

[9] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proc. of the IEEE*, vol.86, no. 4, pp. 615–639, Apr. 1998.

[10] S. Hauck, Z. Li, and E.J. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA," *Proc. FCCM*, pp. 138–146, 1998.

[11] P. Hilton and J. Pederson, "Catalan Numbers, Their Generalization, and Their Uses," *Math. Intelligencer* 13, pp. 64–75, 1991.

[12] A. B. Kahng, "Classical Floorplanning Harmful?" *Proc. of ACM/SIGDA ISPD*, pp. 207-213, April 2000.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp.671–680, May 1983.

[14] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, 1976.

[15] J. Teich, S. P. Fekete, and J. Schepers, "Compile-Time Optimization of Dynamic Hardware Reconfigurations," *Proc PDPTA*, pp. 1097–1103, June 1999.

[16] R. Tesser and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey," *Journal of VLSI Signal Processing*, Vol. 28, no. 1, pp. 7–27, May/June 2001.

[17] Xilinx, "XAPP151 Virtex Series Configuration Architecture User Guide v1.5," Xilinx, Inc., Sep. 2000.

[18] Xilinx "Virtex-II Pro Platform FPGA User Guide," Xilinx, Inc.

[19] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, and H.-L. Chang, "Temporal Floorplanning using 3D-subTCG," *Proc. ASP-DAC*, pp. 725–730, Jan. 2004.