T-Trees: A Tree-Based Representation for Temporal and Three-Dimensional Floorplanning

PING-HUNG YUH and CHIA-LIN YANG
National Taiwan University
and
YAO-WEN CHANG
National Taiwan University

Improving logic capacity by time-sharing, dynamically reconfigurable FPGAs are employed to handle designs of high complexity and functionality. In this article, 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 obtain significantly better solution quality with less execution time than the most recent state-of-the-art work.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: Reconfigurable computing, partially dynamical reconfiguration, temporal floorplanning

A Preliminary version of this article was presented at the *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* 2004 [Yuh et al. 2004a].
This work was partially supported by National Science Council of Taiwan under Grant Nos. NSC 97-2221-E-002-242-MY3, NSC 95-2221-E-002-098-MY3, NSC 96-2221-E-002-250-, NSC 97-2221-E-002-237-MY3, NSC 96-2628-E-002-249-MY3, and NSC 96-2628-E-002-248-MY3, Excellent Research Projects of National Taiwan University, 97R0062-05, ITRI, Springsoft, Synopsys, and TSMC.
Authors' addresses: P.-H. Yuh, C.-L. Yang (contact author), Department of CSIE, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, Taiwan; email: {r91089, yangc}@csie.ntu.edu.tw; Y.-W. Chang, Graduate Institute of EE and Department of EE, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, Taiwan; email: ywchang@cc.ee.ntu.edu.tw.

DOI 10.1145/1562514.1562519 http://doi.acm.org/10.1145/1562514.1562519

ACM Reference Format:
Yuh, P.-H., Yang, C.-L., and Chang, Y.-W. 2009. T-trees: A tree-based representation for temporal and three-dimensional floorplanning. ACM Trans. Des. Autom. Electron. Syst. 14, 4, Article 51 (August 2009), 28 pages.
DOI = 10.1145/1562514.1562519 http://doi.acm.org/10.1145/1562514.1562519

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 AT40KAL series [Atmel 2006], the Altera Stratix II [Altera 2005], the Xilinx Virtex II pro [Xilinx 2005b], and the Virtex 4 [Xilinx 2005a] series. 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 runtime. The capability of runtime reconfiguration makes an FPGA-based design attractive for applications that have different workload at different times, such as wearable computing devices [Plessl et al. 2003], digital signal processing [Schott et al. 2003], mobile portable devices [picochip; quicksilver], and network processors [Chakraborty et al. 2002]. The classic system, which only performs synthesis and optimization in the design time, is unable to meet the required degree of flexibility for the these applications.

A reconfigurable system usually consists a host processor and an FPGA coprocessor called the *Reconfigurable Function Unit (RFU)* [Bazargan et al. 2000]. 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 exist 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.

In this article, we target at the Xilinx Virtex-like FPGA structure. Figure 2(a) shows the Xilinx Virtex model [Xilinx 2000]. The Virtex configuration memory can be considered as an array of bits. The bits of one-bit width that extend

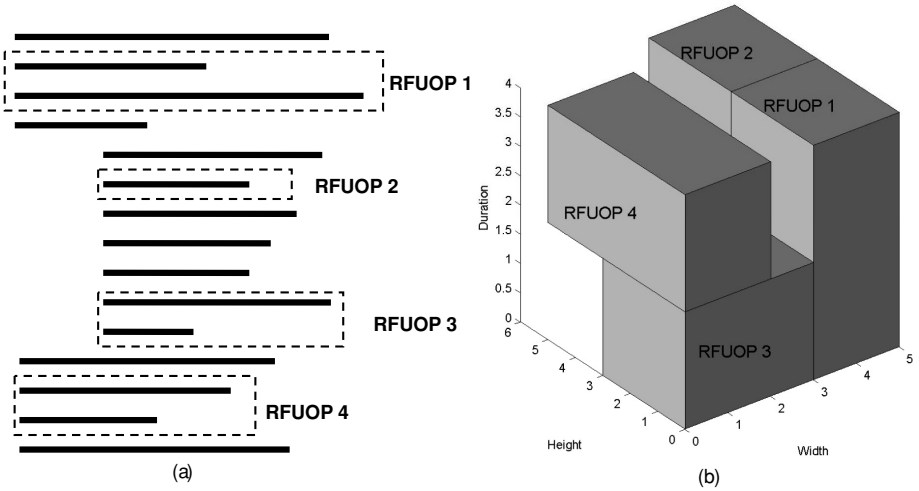

Fig. 1. (a) a running program; (b) a 3D-placement of the running program.

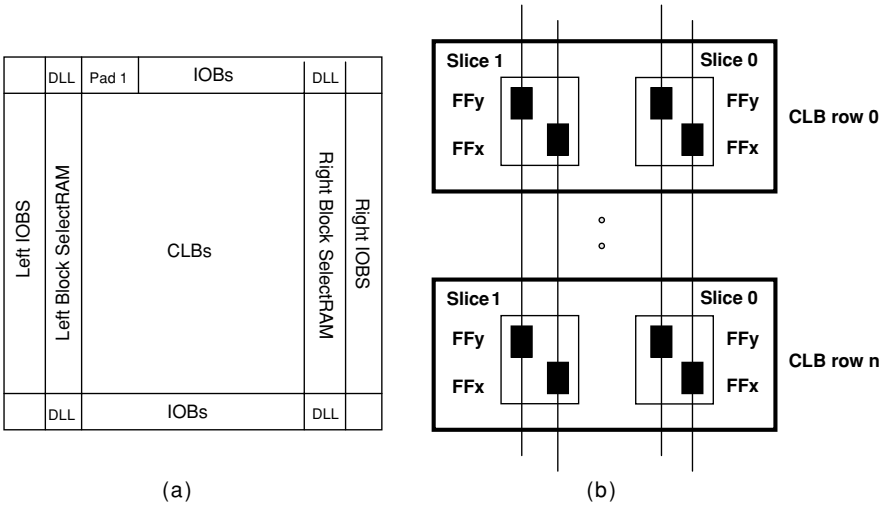

Fig. 2. (a) the Virtex architecture; (b) one column of a 2-slice Virtec CLB.

from the top to the bottom of the array constitute a vertical *frame*, which is the smallest portion of the configuration memory (i.e., the atomic unit that can be written to or read from in this device). Several frames are grouped together into larger units called *columns*. Figure 2(b) shows one column of *Configurable Logic Blocks* (*CLBs* for short). In such a device, we have to specify a full column of a chip for reconfiguration and read-in/out of flip-flop of contents.

One significant purpose of a temporal floorplanner is to be a scheduler. For some applications, the flow of the program is already 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. [1999] 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. [2001] extended their idea and solved the 3D-placement problem with temporal precedence constraints by using component graphs. They apply a search-tree-based method to search for the feasible solutions. The branch of the search tree is done by adding or removing an edge of the component graphs and checking whether the resulting component graphs correspond to a feasible solution. Bazargan et al. in their pioneering work [Bazargan et al. 2000] considered both the offline placement (3D template placement) and the online placement problems. They modeled each RFUOP as a 3D-box and fixed the width and height of the RFU. The major difference between Bazargan et al. [2000] and this article is that they assumed that all RFUOPs are scheduled at compile time while we perform simultaneous scheduling and placement. In the online placement, they dynamically allocated the free space of an RFU to an RFUOP based on different greedy methods (e.g., the best-fit and first-fit heuristics). The goal of the online placement is to find the empty space to accommodate as many RFUOPs as possible. In the offline placement, they proposed a 3D floorplanner which implements four effective methods, including three simulated annealing-based algorithms. The offline placement accepts a solution of the online placement and produces a more compact placement. The work [Ababeii and Bazargan 2006] 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. [2004b] 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 article, 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. In comparison with 3D-subTCG [Yuh et al. 2004b], T-trees have the following advantages.

—Since the operations/perturbations on a 3D-subTCG are performed on edges, its time complexity for operation/perturbations is $O(n^2)$, where n is the number of nodes/modules. In contrast, the operations/perturbations on a T-tree are performed on nodes; therefore, the time complexity for operations/perturbations 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 Yuh et al. [2004b]).

—There are only $O(n!\frac{3^{3n}}{2^{2n}n^{1.5}})$ combinations of a T-tree. Although the authors of 3D-subTCG [Yuh et al. 2004b] do not derive its solution space, we observe that a 3D-subTCG has $\Omega((n!)^3)$ combinations (each transitive closure graph has $\Omega(n!)$ combinations).

To handle the precedence constraints among tasks, we derive an effective and efficient method to examine the feasibility of a T-tree based on the structure of the T-tree; the time of feasibility detection is $O(h)$ 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 reconstructed 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 article 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 both real and synthetic designs. For example, for a large circuit of 300 tasks and 120 precedence constraints, the T-tree-based SA scheme obtains a solution of 17.6% deadspace in 924.34 seconds, while the 3D-subTCG method needs about 6.26 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 article 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 [2000] to address modern floorplanning constraints. Adya and Markov [2003, 2001] 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 article, 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 compared with other methods, such as the offline placement algorithm [Bazargan et al. 2000].

The remainder of this article 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 article, we use task and module interchangeably. Let $V = \{v_1, v_2, \ldots, v_m\}$ be a set of m tasks whose widths, heights, and execution times are given by W_i, H_i, and T_i, $1 \leq i \leq m$. 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, $1 \leq i \leq m$, 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 $D = \{(v_i, v_j) | 1 \leq i, j \leq m, i \neq j\}$ 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 following objectives, that is, volume, wirelength, and reconfiguration overhead. The definitions of these four objective functions are given next.

—*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, that is, 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, these 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-like architecture. Similar to the work by Fekete et al. [2001], 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 [Xilinx 2000]. Therefore, the communication overhead of each reconfiguration takes $360 \times f$

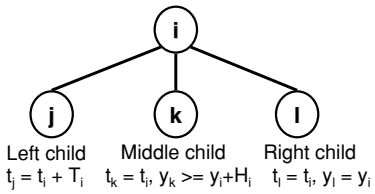

Fig. 3. The structure of T-tree.

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 Xilinx [2000], 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 W_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 article, 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, $1 \leq i \leq m$, such that no two boxes overlap and all precedence constraints are satisfied. The goal of the temporal floorplanning is to optimize a predefined cost metric (defined in the preceding) induced by a placement.

3. THE T-TREE REPRESENTATION

Chang et al. [2000] 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 3. The key insight why we use a ternary tree is that in a 3D space, two adjacent tasks have three possible relations, one in each dimension; that is, task v_j in the X^+ direction of v_i, task v_k in the Y^+ direction of v_i, and task v_l in the T^+ direction of v_i, as shown in Figure 4. Thus, in order to model the relationship among these tasks, we treat v_i as the root and v_j, v_k, and v_l are the children of v_i. Since we have at most three relations in the 3D space, each parent has at most three children.

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, that is, $t_j = t_i + T_i$. 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, that is, $t_k = t_i$ and $y_k \geq y_i + H_i$. If node n_l is the right child of node n_i, module v_l must be placed on the X^+ direction of module v_i, with the t- and y-coordinates equal to those of v_i, that is, $t_l = t_i$ and $y_l = y_i$. Note that there are different ways to define the T-tree structure, and different structures lead to different packing efficiency.

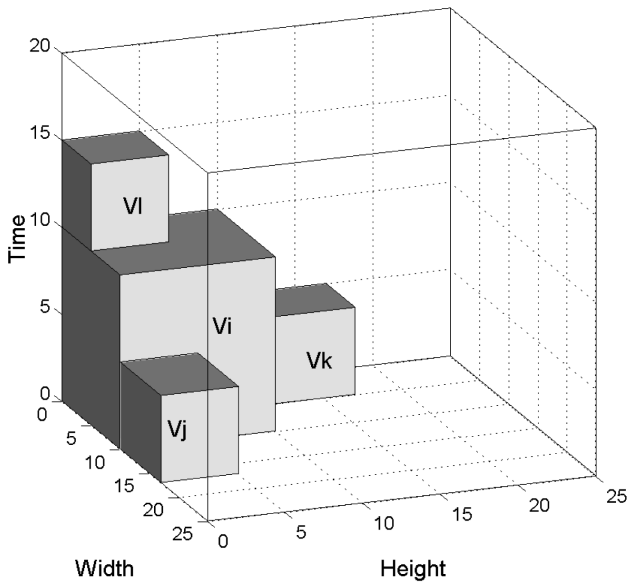

Fig. 4. The three relations in a 3D space.

We will discuss the impact of the tree structures on the packing efficiency in Section 3.3.

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

3.1 From a Compacted Placement to its T-tree

We construct a T-tree for a compacted placement in a Depth-First Search (DFS) manner. The root of the tree corresponds to the task placed at the origin. 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 5 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 construction process takes only linear time.

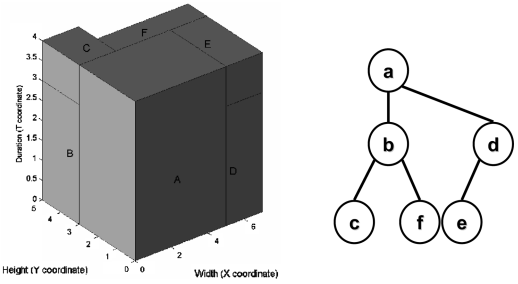

Fig. 5. A compacted placement and the corresponding T-tree.

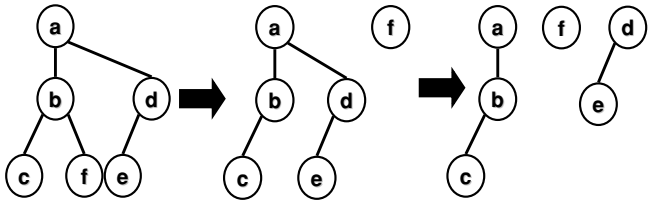

Fig. 6. The T-tree decomposition process.

Based on this construction process, given an acyclic 3D placement, we can represent it with a unique T-tree. An acyclic placement is a compacted placement which does not contain cyclic modules, such as the one shown in Figure 11 where module v_a is adjacent to module v_b in X^+ direction, v_b is adjacent to module v_c in Y^+ direction, and v_c is adjacent to module v_a in T^+ direction.

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 be easily obtained by traversing the T-tree in the DFS order. If node n_j is the left child of node n_i, $t_j = t_i + T_i$; if n_j is the middle or right child of n_i, $t_j = t_i$. Once the t-coordinates are fixed, we can utilize the existing tree solutions in Guo et al. [1999] and Chang et al. [2000] 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 6. 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 6, 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 Guo et al. [1999] and Chang et al. [2000] to determine the y-coordinate of each module. The contour structure is a doubly-linked list of modules that records the contour line in the 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 decomposition process, the packing of each binary tree, and the whole packing procedure are listed in Figure 7, Figure 8, Figure 9, and Figure 10, respectively.

```
Algorithm: Tree Decomposition(n, S)
S: a stack;
n: a node of T-tree;
1 begin
2    stack S';
3    Binary tree B;
4    S'.push(n);
5    B.root = n;
6    while (S' is not empty)
7        node m = S'.pop();
8        if (m.middle_child != NULL)
9            S'.push(m.middle_child);
10           m.b_right_child = m.middle_child;
11       else m.b_right_child = NULL;
12       if (m.left_child != NULL)
13           S'.push(m.left_child);
14           m.b_left_child = m.left_child;
15       else m.b_left_child = NULL;
16       if (m.right_child != NULL)
17           S.push(m);
18   return B;
19 end
```

Fig. 7. Summary of the tree decomposition process.

```
Algorithm: Binary Tree Packing(B)
B: a binary tree;
1 begin
2    stack S;
3    list L;
4    node p = B.root;
5    Place Module(p, L);
6    if (p.right_child != NULL)
7        S.push (p.right_child);
8    if (p.left_child != NULL)
9        S.push (p.left_child);
10   while (S is not empty)
11       p = S.pop();
12       Place Module(p, L);
13       if (p.right_child != NULL)
14           S.push (p.right_child);
15       if (p.left_child != NULL)
16           S.push (p.left_child);
17 end
```

Fig. 8. Summary of the binary tree packing process.

```
Subroutine: Place Module(n, L)
n: a node of a binary tree;
L: a list to store nodes;
1 begin
2    contour = NULL;
3    if (n.parent == NULL)
4        x_n = y_n = t_n = 0;
5        L.insert(n);
6        Update_contour(contour);
7        return;
8    if (n is left child)
9        t_n = t'_{n.parent} + T_{n.parent};
10   else t_n = t'_{n.parent};
11   Find_max_y(contour);
12   Update_contour(contour);
13   L.insert(n);
14   Find_max_x(L);
15 end
```

Fig. 9. The subroutine used in the binary tree packing process.

```
Algorithm: Tree Packing (H)
H: a T-tree;
1 begin
2    stack S;
3    binary tree B;
4    B = Tree Decomposition(H.root, S);
5    Binary Tree Packing(B);
6    while (S is not empty)
7        node n = S.pop();
8        B = Tree Decomposition(n, S);
9        Binary Tree Packing(B);
10 end
```

Fig. 10. Summary of the tree packing process.

The time complexity of the T-tree packing method is $O(n^2)$, which is bounded by the computation of the x coordinates. Although the complexity is the same as the 3D-subTCG [Yuh et al. 2004b], in practice, we observe about 15x speedup over 3D-subTCG when packing 3D-ami49 (the largest circuit used in Yuh et al. [2004b]). Recall that 3D-subTCG uses three transitive closure graphs (one for each dimension) to represent a placement. Therefore, on the average, there are $\frac{n(n-1)}{6}$ edges for one transitive closure graph. The time complexity for computing each coordinate for each task is $O(n^2)$ by applying the well-known *longest path algorithm* [Lawler 1976]. For the T-tree representation, the computation of the t- and y-coordinates takes only $O(n)$ time. Although the computation of the x-coordinate in the T-tree representation takes $O(n^2)$ time, its constant is much smaller compared with that of time complexity of computing the x-coordinate in the 3D-subTCG. Therefore, based on the structure of a T-tree, we are able to develop a more efficient packing method than 3D-subTCG.

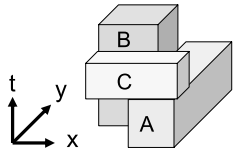

Fig. 11. A 3D placement with three cyclic modules.

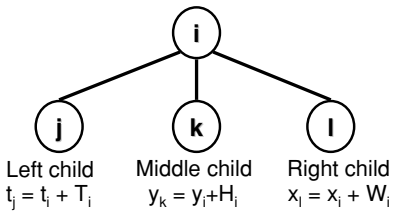

Fig. 12. Another possible structure of T-tree.

THEOREM 3.1. *Given a T-tree, there exists a unique 3D placement corresponding to the T-tree.*

PROOF. Given a T-tree, we can uniquely obtain the coordinates of each task by our packing method. If n_j is the left child of n_i, then $t_j = t_i + T_i$. Otherwise, $t_j = t_i$. The y-coordinate is given by the maximum height of the contour along the T direction in the interval $[t_j, t_j + T_j]$ if n_j is the left or middle child of n_i; otherwise, $y_j = y_i$. For the x-coordinate, we maintain a list L to store all tasks whose y- and t-coordinates are already determined. We set the x-coordinate of v_j as the maximum x'_k for task $v_k \in L$, where the projections of v_k and v_j on T-Y plane are overlapped; that is, $x_j = \max\{x'_k | t_j \le t'_k,\ t_k \le t'_j,\ y_j \le y'_k,\ y_k \le y'_j, v_k \in L\}$. Thus, given a T-tree, we can construct a unique 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.

3.3 Effect of T-tree Structures on Packing Efficiency

As mentioned earlier, there are different ways to define the T-tree structure. Figure 12 shows one possible T-tree structure. Similar to the T-tree structure defined in Figure 3, if n_j is the left (middle/right) child of n_i, task v_j must be placed in the T^+ ($Y^+/X+$) direction of task v_i. Unlike the structure defined in Figure 3, the structure in Figure 12 does not specify the relation between a parent node and its middle/right child nodes in the T^+ direction.

Although the T-tree defined in Figure 12 is more flexible than that of Figure 3, it has higher packing complexity. In this T-tree structure, we cannot decompose a T-tree into a set of binary trees, so we need to directly determine the two coordinates not defined in the structure for each task. It requires to compact each task along its dimension to compute the two coordinates. Therefore, the packing complexity of one task in the structure defined in Figure 12 is increased

from $O(1)$ to $O(n)$ for the y-coordinate (t-coordinate) if this task is the middle (right) child of its parent. Furthermore, the packing complexity of one task is $O(1)$, $O(n)$, $O(n)$ for the x-coordinate, y-coordinate, and t-coordinate if this task is the left child of its parent. Compared with the structure defined in Figure 3, this T-tree structure leads to much higher packing complexity.

4. TEMPORAL FLOORPLANNING ALGORITHM

Our floorplanning algorithm is based on the simulated annealing method [Kirkpatrick et al. 1983]. 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 normalized volume of the placement, W is the normalized total wirelength, O is the reconfiguration overhead, and α, β, and γ are user-specified constants. If a task v_i executes right after a task v_j, the overlapping area of v_i and v_j does not need to be reconfigured. Therefore, the O term is computed based on the following equation. We have

$$O = A(i) + A(j) - OA(i, j), t_i' = t_j, \tag{2}$$

where $A(i)$ is the area of task v_i and $OA(i, j)$ is the overlapping area of tasks v_i and v_j. Note that some existing commercial FPGA architectures, such as the Xilinx Virtex-like architecture, are column- or row-oriented (i.e., configure the whole column/row at a time); our formulation of computing the overhead cell by cell is in fact more general. We can report the width or the height of the overlapping area as the reconfigurable overhead for the Virtex-like architecture. For the communication overhead, after determining the width (number of columns) of each task, we can calculate the communication overhead of each task based on the method presented in Section 2. Then, we add the communication overhead into the execution time of each task. 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 reconstruct the tree if any of the temporal constraints is violated. In what follows, 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 a node can be inserted into: 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. For the internal position, suppose we want to insert node n_i into an internal position between node n_j and node n_k. Without loss of generality, we assume that n_k is the left child of n_j. After inserting n_i in this position, n_i becomes the left child of n_j, and n_k becomes the left child of n_i. For the external position, after inserting n_i to an external position, n_i becomes the left, middle, or right child of some node.

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 task.

4.4 Feasibility Detection and Tree Reconstruction

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 earlier, 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 reconstruct 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 a task v_k. Therefore, the feasibility detection can be summarized in the following statement: *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.*

```
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 ∈ I_k};
6       U = {all nodes in the left subtree of n_p}
7       ∪{n_p};
8       n_j = min{|t_j − t_k| | n_j ∈ U, I_j = ∅};
9       if n_j ≠ n_p then swap nodes n_j and n_k
10      else make n_k the left child of n_p;
11      Calculate t_i for all task v_i based on DFS
12      order of H;
13  until all tasks satisfy the precedence constraint
14  Perform packing for H; // compute the coordinates
15  of all tasks
16 end
```

Fig. 13. Summary of the tree reconstruction process.

Once we identify a node that violates the precedence constraint, we reconstruct 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 $U = \{\text{all nodes in the left subtree of } n_p\} \cup \{n_p\}$. 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 reconstruction process is summarized in Figure 13.

We enhance the tree reconstruction process proposed in Yuh et al. [2004a] by observing that the precedence constraints are only related to the starting time of each task. The physical location of each task does not have any effect on the satisfaction of the precedence constraints. Thus, we do not need to perform packing in the main loop of the tree reconstruction process. Instead, based on the structure of T-tree, we only need to compute the t-coordinate of each task in the tree reconstruction process. After all precedence constraints are satisfied, we perform packing to compute the coordinates of all tasks. We observe 2x speedup for a task graph with 100 tasks and 49 precedence constraints over the original tree reconstruction process.

To avoid infeasible T-trees that may trigger the tree reconstruction process, we develop a mechanism to filter out those operations that will definitely cause precedence constraint violations. Consider the T-tree shown in Figure 14. Assume that 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 DFS order starting *from the right subtree*. For example, in Figure 14, the number in the parentheses of each node denotes the order of the tree traversal.

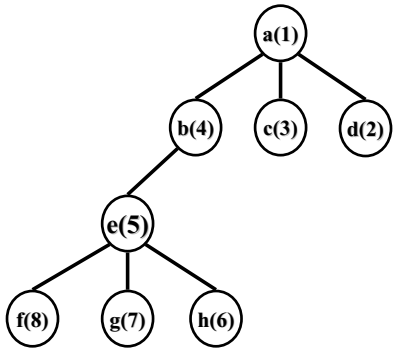

Fig. 14. A T-tree and its DFS order.

In other words, we traverse the T-tree shown in Figure 14 in the following order: n_a, n_d, n_c, n_b, n_e, n_h, n_g, and n_f. The reason why we number each node starting from the right subtree is that we can easily detect if node n_j is in the left subtree of node n_i by observing the DFS numbers of n_j and n_i's left child. If the DFS number of n_j is greater than the DFS number of n_i's left child, then n_j is in the left subtree of n_i. Conventional DFS ordering, that is, traverse the tree starting from the left subtree, does not provide this property. After the numbering, 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 [*DFS_low*, *DFS_up*) based on the previous observation.

4.5 Handling Obstacle Constraint

There might exist regions allocated to static modules, such as IP cores, in an FPGA architecture. To handle this restriction, these static modules are treated as obstacles. Tasks cannot be overlapped with an obstacle. This is called the obstacle constraint. Suppose that $\hat{O}$ represents a set of obstacles. Let $(\hat{x}'_i, \hat{y}'_i)$ be the top-right coordinate of $\hat{o}_i$. During the packing process, we detect if a task v_i overlaps with $\hat{o}_i$. If v_i overlaps with $\hat{o}_i$, we shift v_i to avoid the overlap based on the *X-span* and *Y-span*, which represents how far we should shift v_i along the X (Y) direction to avoid the overlap with $\hat{o}_i$. In this article, s_x (s_y) is the difference between $\hat{x}'_i$ and x_i $(\hat{y}'_i$ and $y_i)$. If s_x is smaller, than we shift v_i in the X direction; otherwise, we shift v_i in the Y direction. Figure 15 summaries how to handle the obstacle constraint.

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 trees and the permutation of n labels. The permutation of n labels is $n!$. From Hilton and Pederson [1991], the counting

Algorithm: Handling Obstacle Constraint (v_i, $\hat{R}$)
v_i: a task;
$\hat{R}$: set of obstacles;
1 **repeat**
2 Scans all obstacles in $\hat{O}$;
3 **if** v_i overlaps with $\hat{o}_j$
4 Let s_x be $|x_i - \hat{x}'_j|$;
5 Let s_y be $|y_i - \hat{y}'_j|$;
6 **if** $s_x < s_y$
7 Set x_i as $\hat{x}'_j$;
8 **else**
9 Set y_i as $\hat{y}'_j$;
10 Update the x-coordinate of v_i;
11 **until** v_i overlaps no obstacles in $\hat{O}$;

Fig. 15. Summary of the obstacle constraint.

of unlabled p-ary tree with n nodes is given by

$$\frac{1}{(p-1)n+1}\binom{pn}{n}. \tag{3}$$

Applying Stirling's approximation, we have

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{4}$$

Setting p equal to 3 in Eq. (3), we can obtain the following asymptotic form.

$$O\left(\frac{3^{3n}}{2^{2n}n^{1.5}}\right) \tag{5}$$

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

$$O\left(n!\frac{3^{3n}}{2^{2n}n^{1.5}}\right). \tag{6}$$

Based on the preceding discussion, we have the following theorem.

THEOREM 5.1. *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 a solution structure, it is possible to find an optimal solution from any initial solution in the solution space. Two placements are said to be *equivalent* if the topologies of their corresponding T-trees, the labeling for each node, and the orientations of all the tasks are the same. In what follows we prove the reachability of the solution space of the T-tree. Note that the reachability described in this subsection is applicable to only nonconstrained cases; that is, there are no precedence constraints.

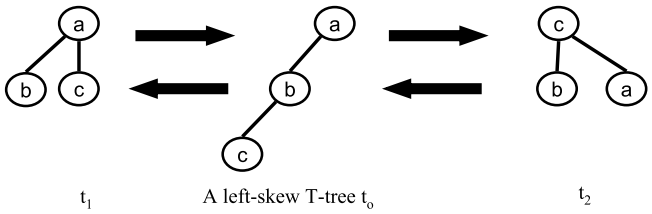

Fig. 16. The transformation process.

THEOREM 5.2. *Given two T-trees H_1 and H_2, H_1 can be transformed to H_2 via at most 4n-3 operations.*

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 16. 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 the 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

$$\Phi' = \alpha V + \beta W + \gamma O + \delta F, \tag{7}$$

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

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

Table I. The 3D-MCNC and 3D-GSRC Benchmarks

Circuit	# of Modules	# of Pads	# of Nets	# of Pins
3D-MCNC Benchmark				
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
3D-GSRC Benchmark				
3D-n100	100	334	885	1873
3D-n200	200	564	1585	3599
3D-n300	300	569	1893	4358

$$(\max(W_p - H_f, 0) \times H_p \times Time) + (\max(H_p - W_f, 0) \times W_p \times Time)), \quad (8)$$

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.

Beside considering the excessive volume in the objective function, we bias the selection of the destination of the Move operations based on the value $p = k/n$, where k is the number of infeasible placements in the last n iterations. In our experiments, we set n equal to 500. A large p 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 p is small, we should try to place a module in the X or Y direction to minimize the task execution time. Based on preceding observation, we modify the probability of choosing the internal and external positions based on p.

7. EXPERIMENTAL RESULTS

Based on simulated annealing [Kirkpatrick et al. 1983], we implemented our T-tree-based temporal floorplanning algorithm in the C++ programming language on a 1.2GHz SUN Blade 2000 machine with 8GB memory. We also implemented the 3D-subTCG-based temporal floorplanner [Yuh et al. 2004b] and the labeled tree- and dual-sequences-based temporal floorplanner [Wang et al. 2008] with the same simulated annealing engine and on the same SUN Blade 2000 machine. All experimental results are the best results obtained by simulated annealing.

7.1 Results on 3D-MCNC and 3D-GSRC Benchmarks

In this section, we first report the results on the 3D-MCNC and 3D-GSRC benchmarks. In this set of experiments, we assume no fixed-outline constraints. Table I summaries the 3D-MCNC and 3D-GSRC benchmarks. The 3D-MCNC benchmark is used in Yuh et al. [2004b]. The 3D-GSRC benchmark is an extension of GSRC by adding task execution time and precedence constraints.

The circuits in the 3D-GSRC benchmark are much larger than those in the 3D-MCNC benchmark.

Table II shows the experimental results of two sets of 3D-MCNC with different numbers of constraints. Note that α and β were all set to 1 for all circuits in order to be consistent with Yuh et al. [2004b]. Compared with the 3D-subTCG, the T-tree can achieve smaller deadspace (14.28% versus 17.85% for the original set of MCNC benchmarks from Yuh et al. [2004b] and 19.02% versus 20.62% for the additional set of MCNC benchmarks) in shorter running time (the time needed for our program to schedule these tasks) (6.21 sec versus 25.69 sec and 12.95 sec versus 37.13 sec). Further, T-tree results in comparable best/average wirelength to 3D-subTCG (e.g., average 416.10 mm versus 323.71 mm for the 3D-MCNC benchmarks, and average 366.62 mm versus 321.9 mm for the 3D-MCNC-2 benchmarks). Figure 17 shows the resulting placement of 3D-ami49 with 11 precedence constraints. The experimental results of 3D-GSRC are summarized in Table III. 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 17.6% deadspace in 924.34 seconds, while the 3D-subTCG method needs about 6.26 hours and results in a solution of 34.2% deadspace.

Finally, we compared with the recent tree-based representation [Wang et al. 2008]. Since they solve only the 3D packing problem, we add extra penalty in the cost function to handle the precedence constraints. The penalty is the time difference of two tasks times the maximum width and height of them if there is a precedence constraint between them and the constraint is violated. Table IV shows the results for 3D-MCNC. As shown in this table, the T-tree representation obtains comparable deadspace (20.0% versus 19.0%) and better wirelength (445.2 mm versus 319.5 mm) in shorter CPU time (14.42 sec versus 6.21 sec). This results show that the T-tree representation is very efficient and effective for the temporal floorplanning problem.

7.2 Results on Synthetic Designs

In this section, we report the results on the synthetic graphs assuming no fixed-outline constraints. In this experiment, we used three sets of synthetic designs generated by TGFF [Dick et al. 1998] to validate our algorithm. TGFF is a task graph generator that can accept various user-specified parameters, such as the maximum in-degree and out-degree of each vertex. We augmented TGFF to randomly generate the width, height, and execution time for each task. We denote three sets of synthetic designs used in the experiment as v25, v50, and v100 (vn represents a graph with n tasks). Table V shows the characteristics of the three sets of synthetic designs. This table lists the ranges of the width, height, and duration of each set of synthetic design. For each design set, we randomly generated five synthetic designs. In this experiment, we applied the enhanced tree reconstruction procedure described in Section 4.4. Since the synthetic designs generated by TGFF do not have the wirelength information, we consider only volume and overhead in this experiment. Table VI shows the result of the synthetic designs. Columns 2 and 3 list the size (the number of vertices)

Table II. Results of Volume and Wirelength Optimization for the Five 3D-MCNC Benchmarks

			3D-subTCG				
Circuit	Total Volume	# of Const.	Width/ Height	Best/avg. Volume	Best/avg. Wirelength (mm)	Best/avg. Dead Space (%)	Best/avg. CPU Time (sec)
3D-apte-1	9.88×10^7	3	1832/3162	1.05×10^8/ 1.22×10^8	244.5/ 341.6	5.9/ 16.8	0.92/ 0.71
3D-xerox-1	4.05×10^7	3	3696/2611	4.82×10^7/ 5.84×10^7	547.3/ 605.1	15.9/ 30.1	0.83/ 0.71
3D-hp-1	1.29×10^7	3	1008/4060	1.63×10^7/ 1.83×10^7	206.2/ 206.3	20.8/ 29.0	1.22/ 1.41
3D-ami33-1	2.32×10^6	7	1120/315	3.17×10^6/ 3.47×10^6	61.2/ 68.2	26.8/ 32.8	44.20/ 78.28
3D-ami49-1	1.32×10^8	11	6342/1708	1.62×10^8/ 1.90×10^8	795.8/ 859.3	18.5/ 30.2	81.37/ 139.90
Average						17.85/ 27.78	25.69/ 44.20
3D-apte-2	9.88×10^7	4	5490/3186	1.04×10^8/ 1.27×10^8	419.4/ 390.0	5.8/ 21.1	0.76/ 0.60
3D-xerox-2	4.05×10^7	4	5355/1316	4.93×10^7/ 6.22×10^7	618.9/ 443.5	17.8/ 34.0	1.40/ 0.84
3D-hp-2	1.29×10^7	6	5096/546	1.66×10^7/ 2.66×10^7	195.9/ 178.5	22.2/ 44.5	1.27/ 0.96
3D-ami33-2	2.32×10^6	15	840/497	3.33×10^6/ 3.97×10^6	63.9/ 63.8	30.3/ 41.2	50.62/ 30.34
3D-ami49-2	1.32×10^8	21	4438/1708	1.81×10^8/ 2.44×10^8	740.2/ 757.3	27.0/ 44.8	122.63/ 59.82
Average						20.62 37.12	37.13 18.51
			T-tree				
Circuit	Total Volume	# of Const.	Width/ Height	Best/avg. Volume	Best/avg. Wirelength (mm)	Best/avg. Dead Space (%)	Best/avg. CPU Time (sec)
3D-apte-1	9.88×10^7	3	1832/3186	1.05×10^8/ 1.14×10^8	240.7/ 350.67	5.9/ 11.83	0.58/ 0.45
3D-xerox-1	4.05×10^7	3	1316/4473	4.70×10^7/ 5.33×10^7	367.1/ 418.3	13.8/ 23.6	0.51/ 0.76
3D-hp-1	1.29×10^7	3	3514/1008	1.41×10^7/ 1.83×10^7	165.1/ 182.4	8.6/ 27.6	0.49/ 0.58
3D-ami33-1	2.32×10^6	7	616/560	3.10×10^6/ 3.38×10^6	56.1/ 49.3	25.1/ 31.1	13.73/ 25.96
3D-ami49-1	1.32×10^8	11	6314/1708	1.61×10^8/ 1.75×10^8	768.6/ 615.2	18.3/ 24.5	15.78/ 45.45
Average						14.28/ 23.72	6.21/ 14.64
3D-apte-2	9.88×10^7	4	5490/3186	1.04×10^8/ 1.19×10^8	388.4/ 368.5	5.8/ 13.6	0.58/ 0. 51
3D-xerox-2	4.05×10^7	4	3591/1316	4.72×10^7/ 5.57×10^7	368.1/ 431.1	14.3/ 26.3	0.90/ 0.55

Continued on next page

Table II. *Continued*

Circuit	Total Volume	# of Const.	3D-subTCG				
			Width/ Height	Best/avg. Volume	Best/avg. Wirelength (mm)	Best/avg. Dead Space (%)	Best/avg. CPU Time (sec)
3D-hp-2	1.29×10^7	6	546/4284	1.63×10^7/ 2.09×10^7	172.1/ 178.3	20.9/ 36.2	0.89/ 0.79
3D-ami33-2	2.32×10^6	15	657/329	3.14×10^8/ 3.38×10^6	48.9/ 53.5	28.8/ 41.1	32.63/ 42.27
3D-ami49-2	1.32×10^8	21	1708/3234	1.76×10^8/ 1.82×10^8	663.5/ 578.1	25.3/ 27.2	29.75/ 52.23
Average						19.02 / 28.88	12.95/ 19.22

(Volume = $mm^2 \times$ clockcycles).

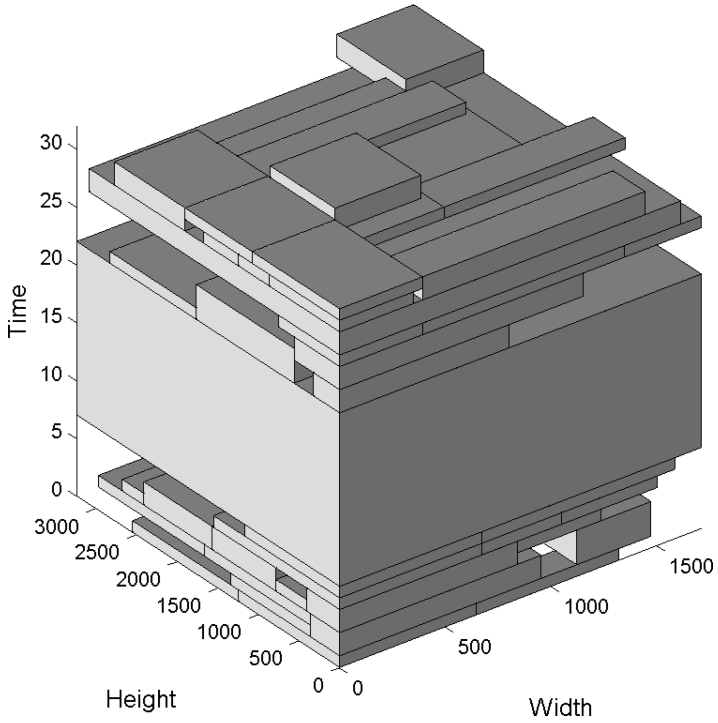

Fig. 17. The result of 3D-ami49-2 with simultaneous volume and wirelength optimization.

and the number of precedence constraints for each synthetic design. Column 4 gives the total volume of each synthetic design. Columns 5 to 10 list the resulting widths, heights, volumes, deadspaces, and CPU times of 3D-subTCG and T-tree. As shown in the table, T-trees can obtain smaller volumes (22.78% deadspace versus 47.07% deadspace) than 3D-subTCG in much less CPU time (39.04 sec versus 140.37 sec). This experiment shows that T-tree is an efficient and effective representation for the 3D scheduling and placement problem.

Table III. Results of Volume and Wirelength Optimization for the Three 3D-GSRC Benchmarks

			3D-subTCG				
Circuit	Total Volume	# of Const.	Width/ Height	Best/avg. Volume	Best/avg. Wirelength (mm)	Best/avg. Dead Space (%)	Best/avg. CPU Time (sec)
3D-n100	5.28×10^6	49	658/67	6.61×10^6/ 9.36×10^6	247.5/ 332.7	20.1/ 34.4	1086.89 852.43
3D-n200	5.27×10^6	88	721/48	6.22×10^6/ 7.97×10^6	540.6/ 697.8	15.3/ 33.0	8658.41 3460.24
3D-n300	8.19×10^6	120	1236/48	1.24×10^7 1.27×10^7	1146.0/ 1136.5	34.2/ 35.8	22553.53 21298.4
Average						23.2/ 33.73	10766.27/ 8203.69
			T-tree				
Circuit	Total Volume	# of Const.	Width/ Height	Best/avg. Volume	Best/avg. Wirelength (mm)	Best/avg. Dead Space (%)	Best/avg. CPU Time (sec)
3D-n100	5.28×10^6	49	168/153	6.16×10^6/ 6.63×10^6	108.7/ 110.5	12.7/ 18.75	62.87/ 64.26
3D-n200	5.27×10^6	88	164/142	6.28×10^6/ 6.70×10^6	231.4/ 227.4	16.2/ 21.3	365.82/ 302.73
3D-n300	8.19×10^6	120	224/185	9.94×10^6/ 1.04×10^7	386.8/ 264.5	17.6/ 21.3	924.34/ 717.02
Average						15.5/ 20.45	451.01/ 568.54

(Volume = $mm^2 \times$ clockcycles).

Table IV. Results of the Comparison with the Labeled and Dual Sequences for the Five 3D-MCNC Benchmarks

	[Wang et al. 2008]			T-tree		
Circuit	Wirelength (mm)	Deadspace (%)	CPU Time (sec)	Wirelength (mm)	Deadspace (%)	CPU Time (sec)
3D-apte	400.2	29.3	0.42	240.7	5.8	0.58
3D-xerox	770.3	11.1	0.38	367.1	14.3	0.51
3D-hp	173.2	15.0	0.38	165.1	20.9	0.49
3D-ami33	58.6	26.7	22.55	56.1	28.8	13.73
3D-ami49	824.1	17.9	48.39	768.6	25.3	15.78
Average	445.2	20.0	14.42	319.5	19.0	6.21

(volume = $mm^2 \times$ clockcycles).

7.3 Results on Real Designs

In this section, we report the results on the five real designs: JPEG encoder [Banerjee et al. 2005a], Recursive Least Square filter (RLS) [TORSCHE], Finite Impulse Filer (FIR), Bandpass Filter (BF) [Papachristou and Konuk 1990], and Fast Fourier Transform (FFT) [Cooly and Tukey 1965]. We assumed that no fixed-outline constraints exist, and considered volume and overhead optimization in this experiment. The width and height of each type of tasks (addition, multiplication, etc.) range from 5 to 15 and the duration ranges from 15 to 25. Table VII shows the result of the five real designs. Columns 2 and 3 list

Table V. Characteristics of the Three Sets of Synthetic Designs

Synthetic Design Set	Width	Height	Exec. Time
v25	[7, 27]	[6, 26]	[18, 22]
v50	[8, 28]	[6, 24]	[32, 48]
v100	[8, 18]	[6, 18]	[38, 62]

Table VI. Results of Volume and Overhead Optimization for the Three Sets of Synthetic Designs

				3D-subTCG			
Circuit	# of Tasks	# of Const.	Total Volume	Width/Height	Volume	Dead Space (%)	Time (sec)
v25_1	25	20	1.3×10^5	55/38	2.1×10^5	35.9	7.26
v25_2	25	24	8.3×10^4	55/36	1.1×10^5	29.7	7.10
v25_3	29	21	3.0×10^5	105/51	4.6×10^5	33.1	8.98
v25_4	27	31	1.3×10^5	85/38	2.6×10^5	47.5	7.51
v25_5	25	20	1.9×10^5	89/45	2.5×10^5	24.1	7.01
v50_1	50	51	4.8×10^5	113/40	9.2×10^5	47.5	42.37
v50_2	52	67	3.1×10^5	80/36	6.5×10^5	51.5	48.63
v50_3	51	48	9.2×10^5	187/50	1.4×10^6	37.7	49.4
v50_4	50	40	5.4×10^5	178/37	8.3×10^5	34.7	44.58
v50_5	49	47	6.3×10^5	117/47	1.2×10^6	48.2	48.63
v100_1	100	118	9.3×10^5	140/45	2.5×10^6	63.8	356.64
v100_2	100	146	4.7×10^5	123/65	1.5×10^6	69.1	321.21
v100_3	102	138	1.3×10^6	170/66	4.0×10^6	65.5	393.49
v100_4	104	112	8.3×10^5	116/55	2.0×10^6	58.5	417.16
v100_5	99	113	8.6×10^5	132/41	2.0×10^6	58.3	345.67
average						47.07	140.37
				T-tree			
Circuit	# of Tasks	# of Const.	Total Volume	Width/Height	Volume	Dead Space (%)	Time (sec)
v25_1	25	20	1.3×10^5	43/23	1.8×10^5	23.9	5.5
v25_2	25	24	8.3×10^4	18/36	1.0×10^5	17.9	1.74
v25_3	29	21	3.0×10^5	26/66	3.6×10^5	16.6	2.28
v25_4	27	31	1.3×10^5	42/22	1.8×10^5	26.2	5.55
v25_5	25	20	1.9×10^5	57/23	2.1×10^5	12.6	4.58
v50_1	50	51	4.8×10^5	38/39	6.5×10^5	25.2	15.99
v50_2	52	67	3.1×10^5	30/20	4.2×10^5	25.4	18.2
v50_3	51	48	9.2×10^5	56/47	1.1×10^6	20.9	9.87
v50_4	50	40	5.4×10^5	56/38	6.9×10^5	21.8	17.46
v50_5	49	47	6.3×10^5	26/42	8.1×10^5	22.5	5.33
v100_1	100	118	9.3×10^5	17/44	1.2×10^6	27.7	85.68
v100_2	100	146	4.7×10^5	26/24	6.5×10^5	27.6	131.03
v100_3	102	138	1.3×10^6	35/35	1.7×10^6	22.3	88.45
v100_4	104	112	8.3×10^5	42/31	1.1×10^6	26.2	102.5
v100_5	99	113	8.6×10^5	32/30	1.1×10^6	25	91.45
average						22.78	39.04

Table VII. Results of Volume and Overhead Optimization for Five Real Designs

Circuit	# of Tasks	# of Const.	Sum of Volume	3D-subTCG			T-tree		
				Volume	Dead Space (%)	Time (sec)	Volume	Dead Space (%)	Time (sec)
JPEG encoder	8	9	17781	25785	31	1.09	25785	31	0.47
RLS	11	12	18448	24150	23.6	11.48	24990	26.2	1.16
FIR	21	12	42672	45824	6.8	3.54	46440	8.1	11.58
BF	29	26	34643	47320	26.7	15.43	46880	26.1	7.46
FFT	64	96	95868	148580	35.4	553.0	142500	32.7	57.03
Average					24.52	116.9		24.64	15.54

Table VIII. Results for Various Aspect Ratios of Desired Widths and Heights for 3D-n100 Circuit

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

the number of tasks and the number of precedence constraints for each circuit, respectively. Column 4 gives the total volume of each circuit. Columns 5 to 10 list the resulting volumes, deadspaces, and CPU times of T-tree and 3D-subTCG. From this table, we can see that T-tree achieves comparable deadspace as 3D-subTCG with significantly shorter CPU time. This experiment confirms our findings in Section 1.2 that T-tree has the advantage of packing efficiency. More importantly, for large-scale circuits such as FFT, T-tree runs one order faster than 3D-subTCG. The experimental result clearly shows that T-tree is more suitable for large-scale circuit designs than 3D-subTCG.

7.4 Results on Fixed-Outline Floorplanning

For the fixed-outline floorplanning problem, we chose the 3D-n100 circuit for experiments. We added various outline constraints. Table VIII reports the success rate[1] and the task execution time[2] of the fixed-outline SA engine described in Section 6. The execution time (clk) listed in this table is total execution time of the scheduled tasks. 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

[1]Number of runs that satisfies the fixed-outline constraint in 100 runs.
[2]The minimum total execution time in all successful runs.

Table IX. Results of the Comparison with Bazargan's Offline Placement Algorithm

Circuit	Outline	[Bazargan et al. 2000]		T-tree	
		Penalty	CPU time (sec)	Penalty	CPU time (sec)
JPEG encoder	15 × 15	271.35	0.16	0	0.13
RLS	20 × 20	2218.11	0.36	0	0.17
FIR	20 × 20	19259.50	0.58	6485.28	0.79
BF	20 × 20	27263.70	0.76	0	2.39
FFT	25 × 25	49250.50	9.33	2245.84	13.93

Penalty is defined as the sum of volumes of all task which cannot be placed within the chip boundary.

the design of the fixed-outline SA engine, we can achieve much higher success rates (88.75% versus 47.37%) at the expense of 5% longer task execution time. The results show the effectiveness of the fixed-outline SA engine.

7.5 Comparison with Bazargan's Offline Placement Algorithm

Finally, in this section, we compare our algorithm with Bazargan's offline placement one [Bazargan et al. 2000]. We used the aforementioned five real designs in this experiment. We report the penalty and CPU time of both algorithms, where penalty is defined as the sum of volumes of all tasks which cannot be placed within the chip boundary [Bazargan et al. 2000]. For our algorithm, we summed up the volumes of all tasks that cannot be placed within the chip boundary. As stated in Section 1.1, Bazargan et al. [2000] assumed that the schedule of each task is performed at compile time. For fair comparison, we first performed floorplanning on the five real designs and then used the resulting schedule as the input to Bazargan et al. [2000]. We used the same low-temperature SA engine for both approaches, and report the results in Table IX based on the average values of 100 SA runs. We can see that our algorithm incurs smaller penalty with longer CPU time, which is a reasonable result since we additionally considered scheduling in our algorithm, and thus our algorithm is more flexible than that in Bazargan et al. [2000].

8. CONCLUDING REMARKS

We have proposed the T-tree representation for handling the temporal (and 3D) floorplanning/placement problem. Compared with the 3D-subTCG, the T-tree can achieve smaller volume and comparable wirelength in less running time. This makes the T-tree particularly suitable for large-scale circuits. The main reasons why the T-tree is more efficient than the 3D-subTCG are threefold: (1) the packing time of T-tree is faster, (2) the size of the solution space is smaller, and (3) the solution space of the T-tree is well structured. As a result, the annealing engine can devote more time to explore the solution space, resulting in smaller volume and less running time.

There is much future work needed to be considered for temporal floorplanning. We give three of them as follows. The first one is the prefetch technique [Hauck 1998] that attempts to hide the significant reconfiguration overhead. This technique decomposes a task into two components, one as the execution part and the other as the reconfiguration part [Banerjee et al. 2005b].

While the execution part of a task is scheduled based on the data dependency (the precedence constraints), the reconfiguration part is not limited by such a dependency. Thus, we can reconfigure a task before it can be scheduled to reduce the reconfiguration overhead. How to incorporate the prefetch technique into the temporal floorplanning will be a great challenge.

The second one is the enhancement of T-trees. Like all other floorplan representations, the T-tree is not perfect (it is arguable if such a perfect one does exist). Since the T-tree is only a partially topological representation, there exist 3D compacted placements that cannot be represented by the T-trees. Based on our empirical study on the T-tree, the incompleteness does not cause problems for practical applications. However, it would be of theoretical interest to close the gap. We are currently investigating techniques to transform a compacted placement that cannot be represented by a T-tree to a compacted placement that can be represented by a T-tree with the same volume.

The third one is the enhancement of T-trees for various floorplanning constraints. The T-tree has the advantage of volume optimization and packing efficiency; however, it may need more effort to handle various floorplanning constraints. For example, Yuh et al. [2007] demonstrated that T-tree may require larger deadspace than 3D-subTCG if the boundary constraints need to be addressed. One possible reason is that T-tree contains less geometrical information than 3D-subTCG. Therefore, the SA engine needs more time to determine if a task is on the four boundaries of a device. Future work also lies in developing an effective data structure to keep more geometrical information while maintaining the same packing time complexity.

REFERENCES

Ababeii, C. and Bazargan, K. 2006. Non-contiguous linear placement for reconfigurable fabrics. *Int. J. Embed. Syst. 2,* 1/2, 86–94.

Adya, S. N. and Markov, I. L. 2001. Fixed-outline floorplanning through better local search. In *Proceedings of the IEEE International Conference of Computer Design*, 328–334.

Adya, S. N. and Markov, I. L. 2003. Fixed-outline floorplanning: Enabling hierarchical design. *IEEE Trans. VLSI Syst. 11,* 6, 1120–1135.

Altera. 2005. *Stratix II Device Handbook*. Altera, Inc.

Atmel. 2006. *AT40K05102040AL*. Atmel.

Banerjee, S., Bozorgzadeh, E., and Dutt, N. 2005a. HW-SW partitioning for architectures with partial dynamic reconfiguration. Tech. rep. CECS-TR-05-02, University of California, Irvine.

Banerjee, S., Bozorgzadeh, E., and Dutt, N. 2005b. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic. In *Proceedings of the Design Automation Conference*, 335–340.

Bazargan, K., Kastner, R., and Sarrafzadeh, M. 2000. Fast template placement for reconfigurable computing systems. *IEEE Des. Test Comput. 17,* 1, 68–83.

B. Schott, Bellows, P., French, M., and Parker, R. 2003. Applications of adaptive computing systems for signal processing challenges. In *Proceedings of the Asia South PAcific Design Automation Conference*, 465–470.

C. Plessl, R. E., Walder, H., Beutel, J., Platzner, M., Thiele, L., and Tröster, G. 2003. The case for reconfigurable hardware in wearable computing. *Personal Ubiq. Comput.* 299–308.

Chakraborty, S., Künzli, M. G. S., and Thiele, L. 2002. Design space exploration of network processor architectures. *Netw. Process. Des. Issues Pract. 1*, 55–89.

Chang, Y.-C., Chang, Y.-W., Wu, G.-M., and Wu, S.-W. 2000. B*-trees: A new representation for non-slicing floorplan. In *Proceedings of the Design Automation Conference*. 458–463.

COOLY, M. AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex fourier series. *Athmatics Comput. 19*, 297–301.

DICK, R. P., RHODES, D. L., AND WOLF, W. 1998. TGFF: Task graph for free. In *Proceedings of International Conference on Hardware Software Codesign*, 97–101.

FEKETE, S. P., KÖHLER, E., AND TEICH, J. 2001. Optimal FPGA module placement with temporal precedence constraints. In *Proceedings of Design, Automation and Test in Eurpoe*, 658–665.

GUO, P.-N., CHENG, C.-K., AND YOSHIMURA, T. 1999. An O-tree representation of non-slicing floorplan and its application. In *Proceedings of the Design Automation Conference*, 268–273.

HAUCK, S. 1998. Configuration pre-fetch for single context reconfigurable processors. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 65–74.

HILTON, P. AND PEDERSON, J. 1991. Catalan numbers, their generalization, and their uses. *Math. Intell. 13,* 2, 64–75.

KAHNG, A. B. 2000. Classical floorplanning harmful? In *Proceedings of the International Symposium on Physical Design*, 207–213.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220,* 4598, 671–680.

LAWLER, E. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston.

PAPACHRISTOU, C. A. AND KONUK, H. 1990. A linear program drive scheduling and allocation method followed by interconnect optimization algorithm. In *Proceedings of the Design Automation Conference*, 77–83.

PICOCHIP. http://www.picochip.com/.

QUICKSILVER. http://www.qstech.com/products.htm.

TEICH, J., FEKETE, S. P., AND SCHEPERS, J. 1999. Compile-time optimization of dynamic hardware reconfigurations. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, 1097–1103.

TORSCHE. *TORSCHE Scheduling Toolbox for Matlab User's Guide v0.2.0b2*.

WANG, R., YOUNG, E. F. Y., ZHU, Y., GRAHAM, F. C., GRAHAM, R., AND CHENG, C.-K. 2008. 3D floorplanning using labeled tree and dual sequences. In *Proceedings of International Symposium on Physical Design*, 54–59.

XILINX. 2000. *XAPP151 Virtex Series Configuration Architecture User Guide v1.5*. Xilinx, Inc.

XILINX. 2005a. *Virtex-4 User Guide*. Xilinx, Inc.

XILINX. 2005b. *Virtex-II Pro and Virtex II Pro X FPGA User Guide*. Xilinx, Inc.

YUH, P.-H., YANG, C.-L., AND CHANG, Y.-W. 2004a. Temporal floorplanning using the T-tree formulation. In *Proceedings of the International Conference on Computer-Aided Design*. 300–305.

YUH, P.-H., YANG, C.-L., AND CHANG, Y.-W. 2007. Temporal floorplanning using the three-dimensional transitive closure subgraph. *ACM Trans. Des. Autom. Electron. Syst. 12,* 4.

YUH, P.-H., YANG, C.-L., CHANG, Y.-W., AND CHANG, H.-L. 2004b. Temporal floorplanning using 3D-subTCG. In *Proceedings of the Asia South Pacific Design Automation Conference*, 725–730.

Received May 2008; revised May 2009; accepted June 2009