Temporal Floorplanning Using the Three-Dimensional Transitive Closure subGraph

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

Improving logic capacity by time-sharing, dynamically reconfigurable Field Gate Programmable Arrays (FPGAs) are employed to handle designs of high complexity and functionality. In this paper, we use a novel graph-based topological floorplan representation, named *3D-subTCG* (3-Dimensional Transitive Closure subGraph), to deal with the 3-dimensional (temporal) floorplanning/placement problem, arising from dynamically reconfigurable FPGAs. The 3D-subTCG uses three transitive closure graphs to model the temporal and spatial relations between modules. We derive the feasibility conditions for the precedence constraints induced by the execution of the dynamically reconfigurable FPGAs. Because the geometric relationship is transparent to the 3D-subTCG and its induced operations (i.e., we can directly detect the relationship between any two tasks from the representation), we can easily detect any violation of the temporal precedence constraints on 3D-subTCG. We also derive important properties of the 3D-subTCG to reduce the solution space and shorten the running time for 3D (temporal) foorplanning/placement. Experimental results show that our 3D-subTCG-based algorithm is very effective and efficient.

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

General Terms: Algorithm, Performance, Design

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

ACM Reference Format:

Yuh, P.-H., Yang, C.-L., and Chang, Y.-W. 2007. Temporal floorplanning using the three-dimensional transitive closure subGraph. ACM Trans. Des. Automat. Electron. Syst., 12, 4, Article 37 (September 2007), 34 pages, DOI = 10.1145/1278349.1278350 http:/doi.acm.org/10.1145/ 1278349.1278350

This work was partially supported by the National Science Council of Taiwan under Grant No's. NSC 94-2220-E-002-001, NSC 95-2752-E-002-008-PAE, NSC 95-2221-E-002-372, and NSC 95-2221-E-002-374 and by the Excellent Research Projects of National Taiwan University, 95R0062-AE00-07.

Author's 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 Electronics Engineering and Department of EE, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, Taiwan; email: ywchang@cc.ee.ntu.edu.tw.

1. INTRODUCTION

An FPGA is a (re)programmable logic device that implements multilevel logic. Traditionally, an FPGA needs to be reconfigured as a whole. Recently, several vendors have proposed architectures that allow partially dynamic reconfiguration, such as the Atmel AT6000 Series FPGAs [Atmel 1997], the Xilinx XC6200 Series FPGAs [Hauck et al. 1998], and the Xilinx Virtex Series FPGAs [Xilinx 2000] [Xilinx]. In the following, we introduce these architectures.

AT6000 Series [Atmel 1997] FPGAs are SRAM-based devices. The important characteristic of this architecture is that we can configure the entire device or selected portions of a design in this architecture. While the portions of the device are configured, others keep operating without disturbing. It takes milliseconds to complete full configuration. Further, partial configuration takes even less time, which depends on its design density.

Figure 1 shows a bitstream file used to configure a hypothetical device, which has 6×6 cells (i.e., number 0 to number 35) surrounded by 16 I/O cells (i.e., number 36 to number 51). According to the bitstream file, it only reconfigures a portion of the array (i.e., the white circle denotes the cells that are reconfigured). The other cells (i.e., the double circles) function as if they are in the normal operational mode.

The XC6200 series FPGAs [Hauck et al. 1998; Xilinx 1996] are also SRAM-based FPGAs. Based on sea of gates, the architecture is hierarchically constructed. The lowest level of the hierarchy lies a large array of simple cells. Each cell can be programmed individually. Besides, the column readins and readouts of flip-flop contents may be performed during runtime without interfering other configured parts of the chip. To make it easier to configure and access the chip's state, all user registers and SRAM control store memory are mapped into a host processor's address space.

To achieve faster configurations in the XC6200 FPGA, it has special hardware called Wildcard Registers, which can be viewed as decompressors for configuration data. The Wildcard Register allows some configuration cells in the same row or column to be written simultaneously. There are two Wildcard Registers, namely Row Wildcard Register and Column Wildcard Register, which are associated with the row address decoder and the column address decoder respectively. Figure 2 shows an XC6216 FPGA with the row decoder and the column decoder. The Wildcard Registers and the address decoders can be viewed as a configuration decompressor. With the decompressor, several cells with the same function can be configured simultaneously. Since the Wildcard Registers can inform the address decoder where locations share the some computation (i.e., they should be configured with the same value), those locations could be configured with a single write operation. However, XC6200 is discontinuous in the middle of 90s.

Figure 3(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 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

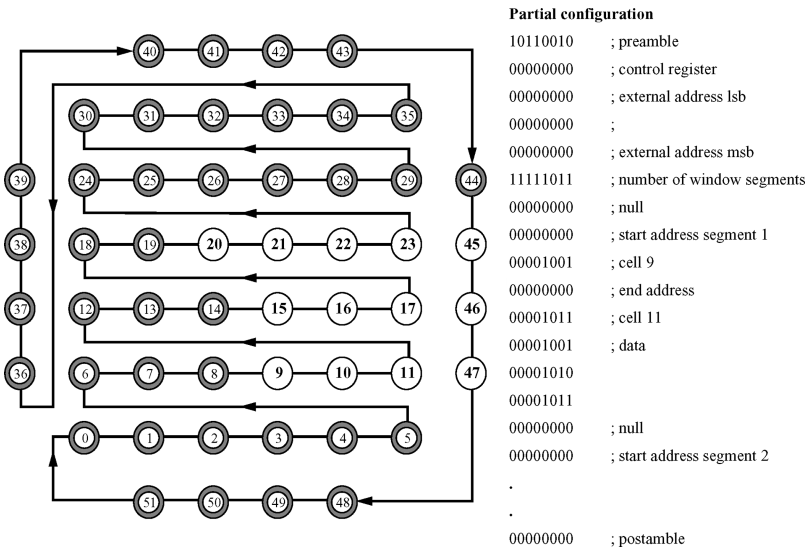

Fig. 1. A partial configuration example of the AT6000 device.

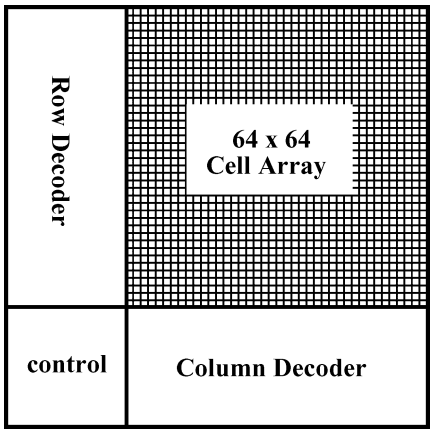

Fig. 2. An Xilinx XC6200 block diagram.

are grouped together into larger units called *columns*. Figure 3(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 readin/out of flip-flop of contents.

Because of the partial reconfiguration capability in an FPGA, studies have shown that an FPGA-based reconfigurable hardware system can improve performance for many applications [Hauck 1998; Tessier and Burleson 2001; Shoa and Shirani 2005]. For example, Chaubal [2004] successfully implemented a partially reconfigurable network controller onto a Xilinx Virtex XCV1000 FPGA device. They also identified three variables that may change during runtime and

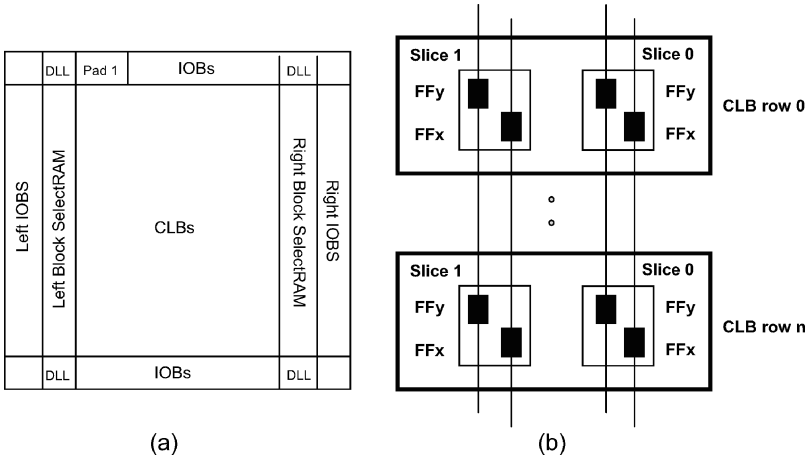

Fig. 3. (a) The Virtex architecture; (b) one-column of a 2-slice Virtex CLB.

thus the partial reconfiguration technique is necessary. These three variables are:

—Number of operational channels.

—Protocol used for a particular channels.

—Parameters associated with the network protocol in use, such as the maximum segment size.

With the feature of partial reconfiguration, we can change the variables of one channel while other channels are not affected. There are two benefits from the partial reconfiguration feature. First, one channel can be reconfigured without affecting any other channels that may be operated in the static part of the design. Second, it reduces the reconfiguration time since the partial bitstream can be significantly smaller than the bitstream for entire design.

Another example is the image interpolation of digital signal processing (DSP) applications. Hudson et al. [1998] developed an image interpolation engine on a Xilinx XC6264 reference board which is consisted of a Xilinx XC6264 FPGA and two on-board memories. The whole design is divided into four stages: Inverse Filter (IF) row computation, IF column computation, Fast Spline Transform (FST) row computation, and FST column computation. The four stages are sequentially loaded into the FPGA. The partial reconfiguration can be used to reduce the reconfiguration overhead because the column and row computation for both the IF and the FST are very similar. They used the *CalDiff* tool [Luk et al. 1997] to generate the partial reconfiguration. By using the partial reconfiguration, the reconfiguration time between the row computation and the column computation is reduced from 300 us to 40 μs, a 99 percent saving in the reconfiguration overhead.

Other researches also demonstrate the benefit of dynamically reconfigurable systems. For example, the wireless communication systems often operates in a time-varying environment. The channel noise characteristic, which depends on environmental parameters, can greatly affect the system performance. Thus,

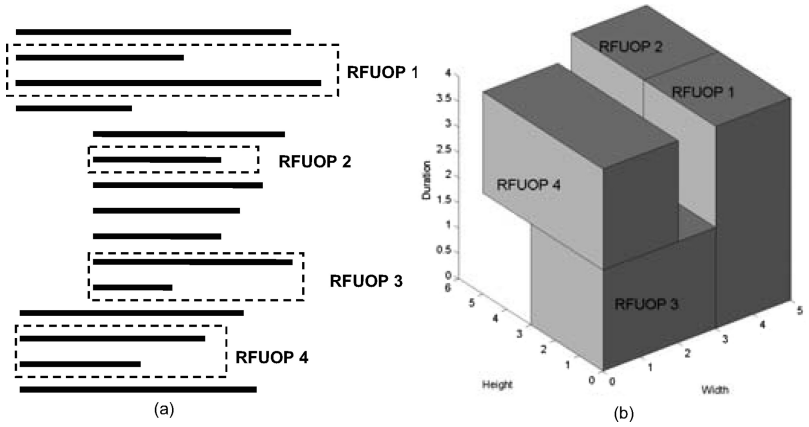

Fig. 4. (a) A running program; (b) a 3D-placement of the running program.

the dynamically reconfigurable hardware can improve the accuracy and the performance of the system based on the runtime conditions. One DSP algorithm which is heavily depend on the runtime condition is the adaptive Viterbi algorithm (AVA). The AVA algorithm can dynamically reduce the average number of computations required per bit of decoded information for comparable bit-error rate based on current channel noise condition [Swaminathan et al. 2002]. If channel noise increases, a more accurate but slower decoder is swapped into the FPGA hardware. Reduced channel noise leads to the opposite effect. Swaminathan et al. [2002] implemented the AVA decoder on a Xilinx XC4036-based PCI board. The overall performance improvement is 7.5X compared with a Celeron processor–based system. For other applications, Alsolaim et al. [2000] achieves more than three times improvement when implementing image processing application onto a dynamically reconfigurable hardware. Durbano and Ortiz [2004] proposed an FPGA-based accelerator to speedup the computational electromagnetic techniques, such as the Finite-Difference Time-Domain method. They achieved up to three times speedup compared with a thirty-node PC cluster. He et al. [2004] proposed an FPGA-based coprocessor for the migration method, which is the most important seismic data processing method. They demonstrated that by integrating the FPGA-based coprocessor with an Intel-based workstation, they can achieve 15.6 times speedup over a 2.4GHz Pentium 4 workstation. Another advantage of an FPGA-based reconfigurable system is that different features can be implemented on the same platform for portable devices [picochip ; quicksilver].

A reconfigurable system is usually composed of a host processor and an FPGA coprocessor, called a *reconfigurable functional unit (RFU)* [Bazargan et al. 2000b]. An RFU, which can be reconfigured during program execution, may have various configurations at different times. Figure 4(a) shows a program with four parts of codes mapped into RFU operations (called *RFUOPs* or *modules*). Because of the area constraint, we may not load all the modules into the device at the same time. Therefore, how to place these modules into the RFU becomes a 3-D placement problem as shown in Figure 4(b). We may

denote each module as a 3-D box with spatial dimensions x and y and the temporal dimension t. Temporal relations exist among scheduled modules since the result of one module may be needed by another one. The objective of temporal floorplanning is either to allocate modules in the RFU to optimize the volume (area times execution time) or to fit all modules within an RFU and minimize execution time, without violating the temporal constraints.

1.1 Previous Work

Teich et al. [1999] first used *component graphs* to handle the temporal floorplanning problem assuming no dependence among scheduled modules. They derived necessary and sufficient conditions for a feasible placement and proposed an enumeration scheme by using a branch-and-bound tree search algorithm to find a feasible solution. In practice, however, there often exist temporal precedence constraints among scheduled modules since the output of one module may be needed as the input of another module. Therefore, Fekete et al. [2001] later extended their work to solve the placement problem with temporal precedence constraints by using an additional dependency graph. Bazargan et al. in their pioneering works [Bazargan et al. 2000b; Bazargan and Sarrafzadeh 1999; Bazargan et al. 2000a] considered both the offline 3D template placement and the online placements. In the offline placement, they modeled each RFUOP as a 3D box and fixed the width and height of an RFU. They proposed a 3D floorplanner that implemented four effective methods, including one greedy method called KAMER-BF (Keep All Maximal Empty Rectangles with Best Fit). In the online placement, they dynamically allocated the free space of an RFU to an RFUOP based on different greedy methods (e.g., best-fit and first-fit). Recently, Yuh et al. [2004] proposed a tree-based representation, called *T-tree*, to handle the temporal floorplanning problem. They used a 3-ary tree to represent the spatial and temporal relationship among RFUOPs. They applied a post-processing step to fix the violations of the precedence constraints after packing. All the aforementioned works assume that the RFUOPs first store results to the external memory and then read back from the external memory. Kaneko et al. [2002] used the Sequence Quintuple representation [Yamazaki et al. 2000] to solve the 3D scheduling problem for dynamically reconfigurable systems. Their formulation included the scheduling and placement of variables between modules. Wu et al. [2001] discussed the placement algorithm if on-chip memory is used for the communications between RFUOPs.

1.2 Our Contribution

In this article, we solve the 3-dimensional floorplanning/placement problem of the general reconfigurable architecture by using a novel topological floorplan representation, called *3D-subTCG* (3-Dimensional Transitive Closure subGraph). To the best knowledge of the authors, this is the first work that uses a graph-based fully topological representation to handle the 3-dimensional placement problem of a dynamically reconfigurable device.

Transitive closure graphs were previously proposed to handle the classical 2D floorplanning/placement problems [Lin and Chang 2001]. The main

challenge to solve the 3D floorplanning problem is that there exist additional *temporal precedence constraints*, for which some tasks must be executed before other tasks start. We use the 3D-subTCG which consists of three transitive closure graphs to model the temporal as well as the spatial relations between tasks/modules. We derive the feasibility conditions for the temporal precedence and the spatial constraints induced by the execution of the dynamically reconfigurable FPGAs. We also derive important properties of the 3D-subTCG to reduce the solution space and shorten the running time for 3D (temporal) foorplanning/placement.

Compared with recently published partial topological representation; that is, *T-tree*, 3D-subTCG has three advantages. First, 3D-subTCG can represent more general 3D floorplans than T-tree. Therefore, the solution space of T-tree may not contain the optimal solution when considering other objectives, such as the minimization of wirelength. Second, the geometric relationship is transparent to both the representation and the induced operation; that is, we can detect the relationship between any two tasks directly from the representation. We can easily detect any violation of temporal precedence and spatial constraints in the 3D-subTCG. Therefore, we can guarantee a feasible solution without resorting to time-consuming postprocessing to remove infeasible ones. Third, since the geometric relations among tasks can be directly obtained from the representation, 3D-subTCG may be more suitable for handling various practical placement constraints. For example, some boundary modules are desired to be placed on the boundary of a reconfigurable device for shorter connection to I/O blocks.

In this article, we consider two floorplanning problems. The first one is the classical temporal floorplanning problem that minimizes the product of the area and the execution time (i.e., the volume of the 3D floorplan/placement). The volume optimization is to analyze the tradeoff between the required area and the total execution time. Experimental results show that our 3D-subTCG-based algorithm can obtain significantly better floorplans than the Sequence Triplet (ST) representation [Yamazaki et al. 2000] (35.18% deadspace in ST vs. 14.86% in 3D-subTCG). The running-time requirement of 3D-subTCG is also significantly smaller than ST (312.18 sec as ST vs. 166.46 sec as 3D-subTCG).

The second one is the temporal floorplanning with constraints. First, we extend our 3D-subTCG-based algorithm to handle the temporal floorplanning with placement constraints (e.g., boundary modules). The experimental result shows that 3D-subTCG can achieve shorter average wirelength and smaller average deadspace compared with the T-tree representation. Then, we 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 [2001, 2003] 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 their idea to propose a new cost function to guide the simulated annealing engine. The experimental results show that our fixed-outline temporal floorplanner is effective of fitting 3D-blocks into the desired outline.

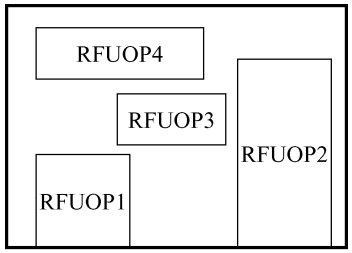

Fig. 5. The flexible 2D reconfigurable area model.

The remainder of this article is organized as follows. Section 2 formulates the temporal floorplanning problem. Section 3 reviews the Transitive Closure Graph (TCG) representation and presents the 3D-subTCG for temporal floorplanning. Section 4 introduces our temporal floorplanning algorithm. Section 5 extends our 3D-subTCG-based algorithm to handle various constraints. Section 6 reports the experimental results. Finally, conclusions are given in Section 7.

2. FORMULATION

In this section, we introduce the problem formulation of temporal floorplanning. First we describe the RFU architecture used in this paper. The RFU architecture assumed in this paper consists of a set of basic cells arranged in a 2D area array. The basic cell is the smallest unit that an RFUOP can be allocated. Each basic cell can be allocated to at most one RFUOP at one time. We assume that the 2D flexible area model is used, where each RFUOP can be freely allocated anywhere in an RFU. Figure 5 shows the area model. In addition, we assume that an RFUOP can be reconfigured at any time. This assumption may not match current RFU architectures, where we can only reconfigure an RFUOP at one time. For RFUOPs, we assume that each RFUOP is a rectangle and the execution time can be determined in advance. We also assume that the RFUOPs are nonpreemptive.

In the reconfigurable architecture, a *task* v is loaded into the device for a period of time for execution. Let $V = \{v_1, v_2, \ldots, v_m\}$ be a set of m tasks whose widths, heights, and durations are denoted by W_i, H_i, and T_i, $1 \leq i \leq m$. Let (x_i, y_i) denote the coordinate of the bottom-left corner of a task v_i on the chip, where $1 \leq i \leq m$.

To guarantee the correctness of the functions in the reconfigurable architecture, we must satisfy the temporal precedence requirements, which describe the temporal ordering among tasks. We refer to the temporal precedence requirements as the *precedence constraints*. Let $D = \{(v_i, v_j) | 1 \leq i, j \leq m, i \neq j\}$ denote the precedence constraints for the tasks v_i and 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 four objective functions:

—*Volume (the minimum bounding box of a placement).* In temporal floorplanning, we need to consider the tradeoff 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 time if a smaller one is used. Therefore, we shall minimize the product of the area of the device and the total execution time.

—*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 problem is different from the traditional floorplanning/placement problem. Given a net, those nodes in the net may be executed at the same time frame or at different time frames. If they are executed at the same time frame, we can estimate the wirelength according to their geometric distance directly. However, we have to project all nodes into the same time frame before computing their wirelength in the other condition.

—*Communication overhead.* We quantify the communication overhead based on the Xilinx Virtex architecture described in Section 1. Similar to 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 successor task, are first written to the external memory through the bus interface. The dependent task, which has been loaded at the specified position, then performs 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 only 8 bit. Thus, it takes approximately $1248/8 + 24 = 180$ clock cycles in each readin or readout, where the 24 cycles are the configuration overhead of the bus interface as described on the Xilinx FPGA data book [Xilinx 2000]. Therefore, the communication overhead is $360 \times f$ 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 Section 1, Xilinx Virtex architecture is column-oriented (i.e., all bits in one column should be updated in each readin or readout). 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 per CLB column. This means that we need $W_i \times 7512$ clock cycles in total if the addresses in the column are incrementally updated.

In this paper, we treat a task v_i as a three-dimensional box. A placement $\mathcal{P}$ is an assignment of (x_i, y_i, t_i) for each v_i, $1 \leq i \leq m$, where t_i is the starting time for the scheduled task, such that no two boxes overlap and all precedence constraints are satisfied. The goal of temporal floorplanning is to optimize a set of predefined cost metrics induced by a placement.

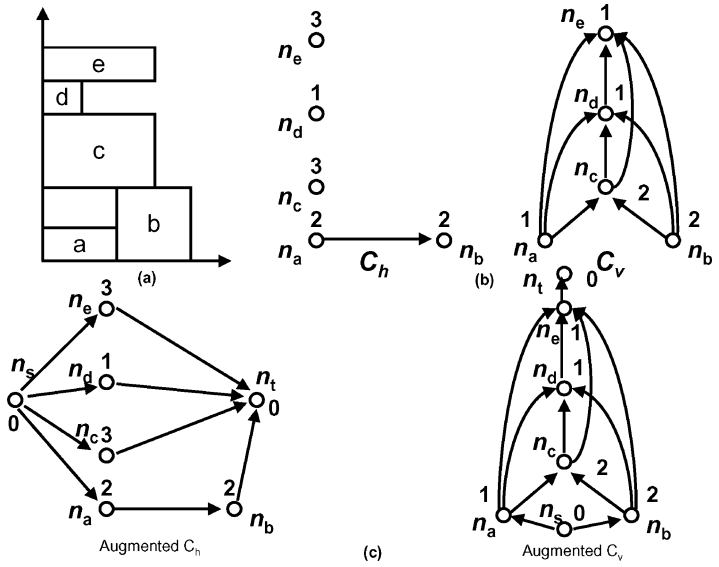

Fig. 6. (a) A placement; (b) TCG; (c) augmented TCG (augmented C_h and C_v).

3. 3D-SUBTCG FOR TEMPORAL FLOORPLANNING

3.1 Review of TCG

We first review the Transitive Closure Graph (TCG) representation presented in Lin and Chang [2001]. TCG uses two graphs, *a horizontal transitive closure graph* C_h and *a vertical transitive closure graph* C_v, to describe the geometric relations among modules. For two nonoverlapping modules b_i and b_j, b_i is said to be *horizontally* (*vertically*) *related* to b_j, denoted by $b_i \vdash b_j$ ($b_i \perp b_j$), if b_i is on the left (bottom) side of b_j and their projections on $y(x)$ axis overlap. For two nonoverlapping modules b_i and b_j, b_i is said to be *diagonally related* to b_j if b_i is on the left side of b_j, and their projections on the x axis and y axis do not overlap. To simplify the operations on geometric relations, we treat a diagonal relation as a horizontal one, unless there exist a chain of vertical relations from b_i (b_j), followed by the modules enclosed with the rectangle defined by the two closest corners of b_i and b_j, and finally to b_j (b_i), for which we make $b_i \perp b_j (b_j \perp b_i)$. For each module b_i, we introduce one node n_i both in C_h and C_v. If $b_i \vdash b_j$, a directed edge (n_i, n_j) is constructed in C_h. Similarly, we construct a directed edge (n_i, n_j) in C_v if $b_i \perp b_j$. Figure 6(a) shows a placement with five modules a, b, c, d, and e whose widths and heights are (2, 1), (2, 2), (3, 2), (1, 2), and (3, 1), respectively. Figure 6(b) shows the TCG corresponding to the placement of Figure 6(a). The weight of each node in C_h (C_v) represents the width (height) of the corresponding module b_i. Since $b_a \vdash b_b$, we construct a directed edge (n_a, n_b) in C_h. Similarly, since $b_a \perp b_c$, a directed edge (n_a, n_c) is constructed in C_v.

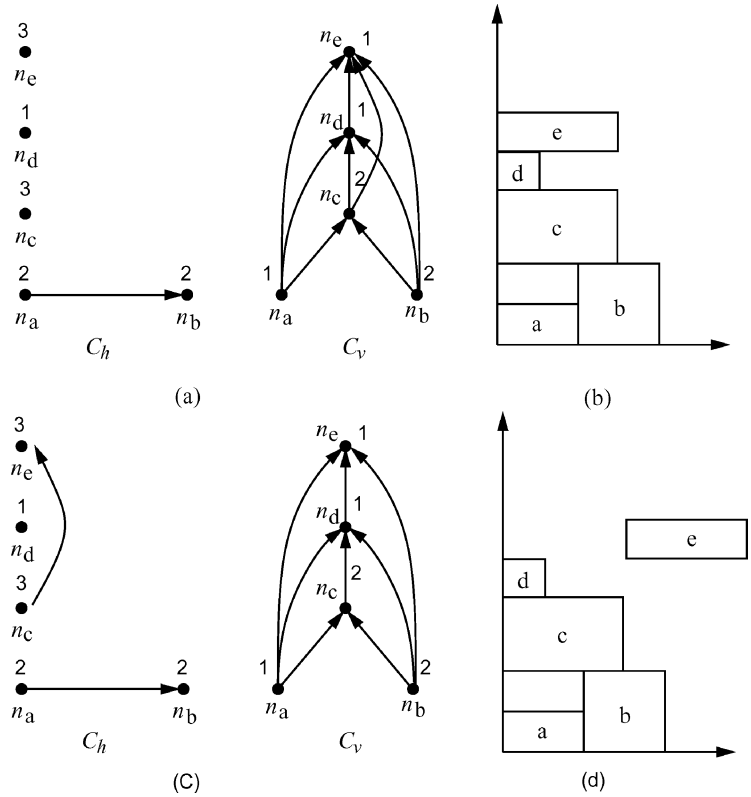

Fig. 7. (a) A feasible TCG that the edge (n_c, n_e) lies in C_v; (b) the corresponding placement of Figure 7 (a); (c) a redundant TCG that the edge (n_c, n_e) lies in C_h; (d) the corresponding placement of Figure 7 (c).

TCG has the following three *feasibility properties* [Lin and Chang 2001]:

(1) C_h and C_v are acyclic.
(2) Each pair of nodes must be connected by exactly one edge either in C_h or in C_v.
(3) The transitive closure of C_h (C_v) is equal to C_h (C_v) itself.[1]

The first property ensures that a module b_i cannot be both left and right to (below and above) another module b_j in a placement. The second property guarantees that no two modules overlap since each pair of modules have exactly one of the horizontal or vertical relation. The third property eliminates redundant solutions. Figure 7 illustrates the third property. As shown in Figure 7(a), since there is a path from node n_c to node n_e in C_v, the edge (n_c, n_e) must be in C_v. If we place the edge (n_c, n_e) into C_h, as shown in Figure 7(c), the resulting area of the placement must be larger or equal to the configuration of Figure 7(a).

[1]The transitive closure of a directed acyclic graph G is defined as the graph $G' = (V, E')$, where $E' = \{(n_i, n_j)$: there is a path from node n_i to node n_j in $G\}$.

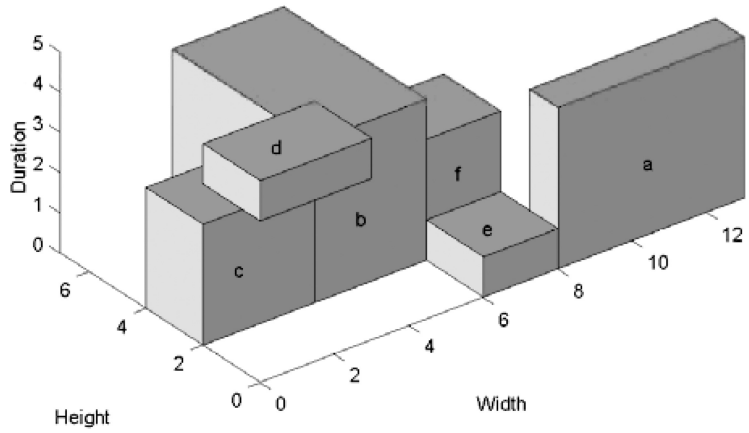

Fig. 8. A placement.

Figure 7(b) and 7(d) shows the two placements. The third property eliminates this redundant solution.

Given a TCG, a placement can be obtained in $O(m^2)$ time by performing a well-known *longest path algorithm* [Lawler 1976] on TCG, where m is the number of modules. To facilitate the implementation of the longest path algorithm, the two closure graphs can be augmented as follows. For each closure graph, we introduce two special nodes with zero weights, the source n_s and the sink n_t, and construct an edge from n_s to each node with in-degree equal to zero and also from each node with out-degree equal to zero to n_t. Figure 6(c) shows the augmented TCG for the TCG shown in Figure 6(b).

Let $L_h(n_i)$ ($L_v(n_i)$) denote the weight of the longest path from n_s to n_i in the augmented C_h (C_v). $L_h(n_i)$ ($L_v(n_i)$) can be determined by performing the single source longest path algorithm on the augmented C_h (C_v) in $O(m^2)$ time, where m is the number of modules. The coordinate $(x_i,\ y_i)$ of a module b_i is given by $(L_h(n_i),\ L_v(n_i))$. Further, the coordinates of all modules are determined in the topological order in C_h (C_v). Since the respective width and height of the placement for the given TCG are $L_h(n_t)$ and $L_v(n_t)$, the area of the placement is given by $L_h(n_t) \times L_v(n_t)$. Since each module has a unique coordinate after packing, there exists a unique TCG corresponding to any placement.

3.2 3D-subTCG

As shown in the previous subsection, TCG describes the geometric relations among modules based on two graphs, C_h and C_v. For a dynamically reconfigurable device, there exists the temporal ordering among tasks. For two tasks v_i and v_j, v_i is said to be *temporally related* to v_j, denoted by $v_i \prec v_j$, if v_i is executed before v_j starts. Note that $\prec$ only states the execution order among tasks. It does not define the precedence constraint. To solve the 3D floorplanning/placement problem, we need to consider the temporal and spatial relations at the same time. Therefore, we introduce a new graph to model the temporal

Table I. The Coordinates, Width, Height, and Duration for Each Task in Figure 8

Task	Coordinates (x_i, y_i, t_i)	Width	Height	Duration
a	(8,0,0)	5	1	4
b	(2,2,0)	3	5	4
c	(0,2,0)	3	2	3
d	(0,0,4)	3	2	1
e	(6,0,0)	2	2	1
f	(6,2,1)	2	2	3

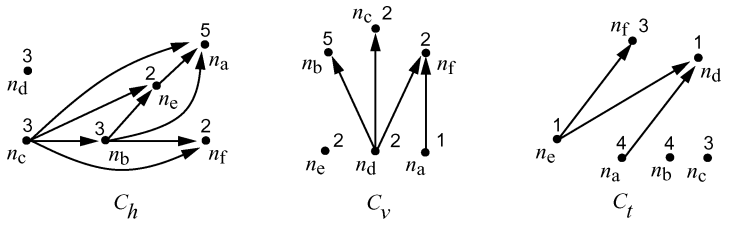

Fig. 9. The corresponding 3D-subTCG of Figure 8.

relations among tasks, namely a *temporal transitive closure graph* C_t. This new representation is called 3D-subTCG, which contains three transitive graphs, C_h, C_v, and C_t. For each task v_i, we construct one node n_i in each graph. If $v_i \vdash v_j$ ($v_i \perp n_j$), we construct an edge (n_i, n_j) in C_h (C_v). If v_i is executed before v_j starts, we construct an edge (n_i, n_j) in C_t.

Figure 8 shows a placement with six tasks a, b, c, d, e, and f. Table I shows the coordinate, width, height, and duration of each task in Figure 8. Figure 9 shows the 3D-subTCG corresponding to the placement of Figure 8. The value associated with a node in C_h (C_v or C_t) gives the width (height or duration) of the corresponding task, and the edge (n_i, n_j) in C_h (C_v or C_t) represents the horizontal (vertical or temporal) relation of v_i and v_j. In Figure 9, since task v_c (v_a) is left to (below) v_b (v_f), there exists an edge (n_c, n_b) ((n_a, n_f)) in C_h (C_v). Similarly, since task v_a must be executed before task v_d, there exists an edge (n_a, n_d) in C_t.

To obtain the coordinate of each task, we apply the longest path algorithm to the three graphs in a 3D-subTCG. (See Section 3.1 for the details.)

3D-subTCG has the following three *feasibility properties*:

(1) C_h, C_v, and C_t are acyclic.
(2) Each pair of nodes must have exactly one edge in either C_h, C_v, or C_t.
(3) There must exist an edge (n_i, n_j) if there is a path from n_i to n_j in one graph and there exist no edges between n_i and n_j in other graphs.

The first two properties, which are the same as TCG, guarantee that a solution is feasible. The third property is to eliminate the redundant solutions. An edge (n_i, n_j) is said to be a *closure edge* if there exists a path from node n_i to node n_j except the edge (n_i, n_j) itself. For example, the edges (n_b, n_a), (n_c, n_a), (n_c, n_e),

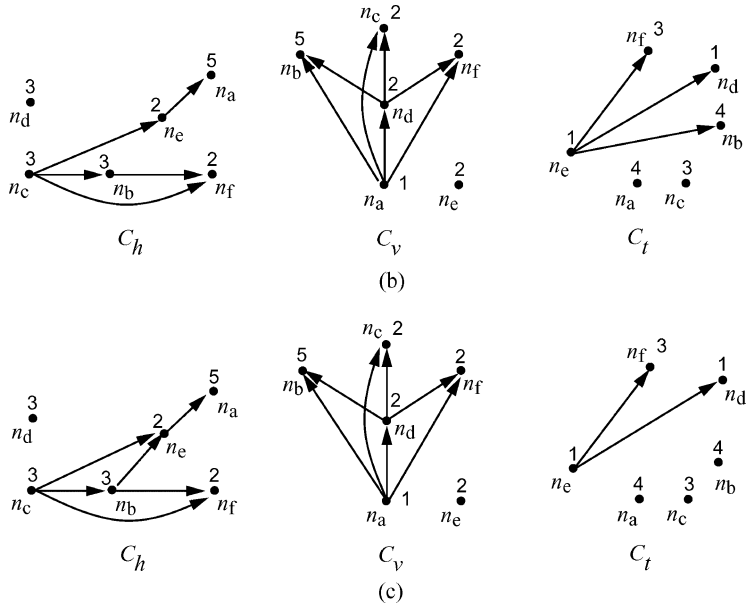

Fig. 10. (a) A 3D-subTCG with only one path between node n_a and n_b in C_v; (b) a 3D-subTCG contains two paths in C_h and C_v between node n_a and n_b.

and (n_c, n_f) in C_h of Figure 9 are closure edges. If there exists a path from node n_i to node n_j in one graph, the closure edge (n_i, n_j) should appear in the same graph instead of others to eliminate the redundant solutions as explained in Section 3.1. However, before adding a new closure edge (n_i, n_j) after each operation, we need to make sure that there exist no closure edges between n_i and n_j in other graphs. Figure 10 illustrates this scenario. Figure 10(a) shows a 3D-subTCG that there exists a closure edge (n_a, n_b) between two nodes n_a and n_b. Figure 10(b) shows the resulting graph after deleting the edge (n_e, n_b) in C_t and adding the edge (n_b, n_e) to C_h. Now there is a path from n_b to n_a in C_h. However, in order to maintain the second property, we cannot add the closure edge (n_b, n_a) in C_h since (n_a, n_b) has already existed in C_v.

3.3 Discussions

There are other works that also use the topological representation, that is, *T-tree* [Yuh et al. 2004], to solve the temporal floorplanning problem. In this subsection, we discuss the pros and cons of 3D-subTCG and T-tree representations.

Although the T-tree representation outperforms the 3D-subTCG in terms of packing efficiency and volume optimization, 3D-subTCG have the following three advantages over T-tree:

—3D-subTCG is a fully topological representation that can represent the general topological modeling of tasks, and thus contains a complete solution structure for searching the optimal floorplan/placement solution. In contrast, T-tree is a partially topological representation and can only represent the

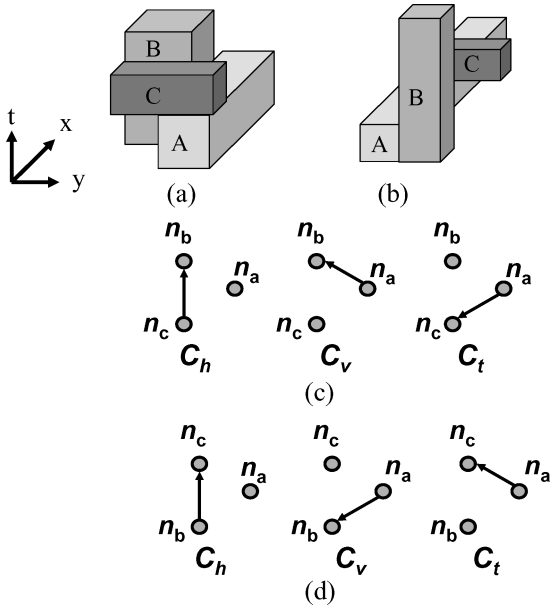

Fig. 11. Compacted floorplans that can be represented by 3D-subTCGs, but not T-trees. (a) A compacted floorplan with no task at the origin; the floorplan cannot be modeled by a T-tree; (b) a compacted floorplan that cannot be represented by a T-tree; (c) the corresponding 3D-subTCG of (a); (d) the corresponding 3D-subTCG of (b).

compacted 3D floorplans where each task must be compacted to the origin. Furthermore, there exist 3D compacted floorplans that cannot be represented by a T-tree but can be represented by a 3D-subTCG. Figure 11 shows two compacted 3D floorplans that cannot be represented by a T-tree. Figure 11(a) shows a 3D floorplan with no task at the origin. Since T-tree requires a root node corresponding to the task at the origin, this 3D floorplan cannot be represented by a T-tree. Figure 11(b) shows another 3D floorplan with a task at the origin. This 3D floorplan has three direct relations; that is, task v_b is in the Y^+ direction of task v_a and is adjacent to v_a, task v_c is in the T^+ direction of task v_a and is adjacent to v_a, and task v_c is in the X^+ direction of task v_b and is adjacent to v_b. This 3D floorplan cannot be represented by a T-tree because the packing method and the tree structure proposed in Yuh et al. [2004] cannot generate a 3D floorplan with three direct relations. However, both 3D floorplans can be easily represented by a 3D-subTCG. Figure 11(c) shows the corresponding 3D-subTCG of Figure 11(a), and Figure 11(d) shows the corresponding 3D-subTCG of Figure 11(b).

Moreover, the incomplete solution space of T-tree may not contain the optimal solution when considering other optimization objectives, such as the wirelength minimization. Figure 12 shows two 3D floorplans. Suppose there are strong interconnections between tasks v_a and v_c. The wirelength of Figure 12(a) is smaller than the wirelength of Figure 12(b) because v_a is adjacent to v_c in Figure 12(a). Since Figure 12(a) is not a compacted placement, a T-tree cannot represent Figure 12(a) and can only represent Figure 12(b).

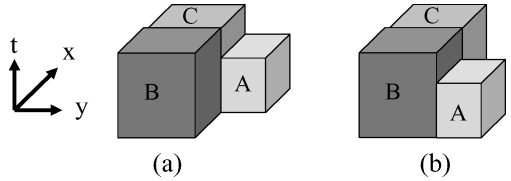

Fig. 12. (a) A floorplan that can be represented by a 3D-subTCG; (b) the compacted floorplan resulted from the T-tree packing. Box A is packed to the position with $x = 0$.

Thus, T-tree may lose the optimality when considering other objectives.

—Because the relation between each pair of tasks is defined in the representation, the geometric relation of each pair of tasks is transparent to both the representation and the induced operations; that is, we know the change of the geometric relations among tasks *before* perturbation. Thus, we can perform the feasibility detection *before* perturbation to guarantee the satisfaction of precedence constraints. In contrast, T-tree is a partially topological representation where some geometric relations among tasks cannot be obtained directly from representation. Thus, it is hard to detect the violations of the precedence constraints before packing and a post-processing is required to guarantee the feasibility of the solutions after packing.

—Since the geometric relations among tasks can be directly obtained from the representation, 3D-subTCG may be more suitable for handling various practical placement constraints. For example, since the input/output blocks are on the boundary of the reconfigurable devices, such as the Xilinx Virtex shown in Figure 3, some tasks are desired to be placed on the boundary of a device. We can easily detect if a task is on the boundary of the device by observing the in-degree/out-degree of its corresponding node in C_h or C_v. We can also detect if a task starts at time step zero on an RFU by observing the in-degree/out-degree of its corresponding node in C_t. As shown in Figure 9, a node with zero in-degree (out-degree) in C_h corresponds to a task on the left (right) boundary. Similarly, a node with zero in-degree (out-degree) in C_v corresponds to a task on the bottom (top) boundary. If a node with zero in-degree in C_t, then the corresponding task starts at time zero. By observing the in-degree/out-degree of a node n_i in C_h and C_v, we can easily detect if task v_i is on the boundary of a device.

4. TEMPORAL FLOORPLANNING ALGORITHM

Our algorithm is based on simulated annealing [Kirkpatrick et al. 1983]. Given an initial 3D-subTCG, we perturb the 3D-subTCG to obtain a new 3D-subTCG. The cost function Φ used in our algorithm is given by

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

where V is the volume of the placement, W is the 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:

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

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 (the 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. In this section, we first describe how to identify a reduction edge, and then show the perturbation operations in simulated annealing. Then, we introduce the feasibility condition that a 3D-subTCG must satisfy during each perturbation in order to maintain the correct temporal ordering among tasks.

4.1 Reduction Edge Identification

First we illustrate the concept of *reduction edges*. An edge (n_i, n_j) is called a *reduction edge* if there does not exist another path from node n_i to node n_j except the edge (n_i, n_j) itself. For example, the edges (n_b, n_f), (n_b, n_e), and (n_e, n_a) in C_h of Figure 9 are reduction edges while edges (n_c, n_f), (n_c, n_a), and (n_b, n_a) are not. Recall that 3D-subTCG is formed by directed acyclic transitive closure graphs. Given an arbitrary node n_i in one transitive closure graph, there exists at least one reduction edge (n_i, n_j), where $n_j \in F_{out}(n_i)$. Here we define the fan-in (fan-out) of a node n_i, denoted by $F_{in}(n_i)$ ($F_{out}(n_i)$), as the nodes n_j's with edges (n_j, n_i) ((n_i, n_j)). For nodes $n_k, n_l \in F_{out}(n_i)$, the edge (n_i, n_k) cannot be a reduction edge if $n_k \in F_{out}(n_l)$. Hence, we remove those nodes in $F_{out}(n_i)$ that are fan-outs of others. The edges between n_i and the remaining nodes in $F_{out}(n_i)$ are reduction edges. In the C_h of Figure 9, $F_{out}(n_c) = \{n_a, n_b, n_e, n_f\}$. Since n_a, n_e, and n_f belong to $F_{out}(n_b)$, edges (n_c, n_a) and (n_c, n_f) are closure edges while (n_c, n_b) is a reduction one. The reason for identifying reduction edges is that the operations defined below are only applied to reduction edges. The time complexity of finding such a reduction edge is $O(m^2)$, where m is the number of tasks (tasks) [Lin and Chang 2001].

4.2 Solution Perturbation

We define the following five operations to perturb a 3D-subTCG:

—*Rotation.* Rotate a task.
—*Swap.* Swap two nodes in C_h, C_v, *and* C_t.
—*Reverse.* Reverse a *reduction edge* in C_h, C_v, *or* C_t.
—*Move.* Move a *reduction edge* from one graph (C_h, C_v, *or* C_t) to another graph.
—*Transpositional Move.* Move a *reduction edge* from one graph (C_h, C_v, *or* C_t) to another graph, and then transpose the two nodes associated with the edge.

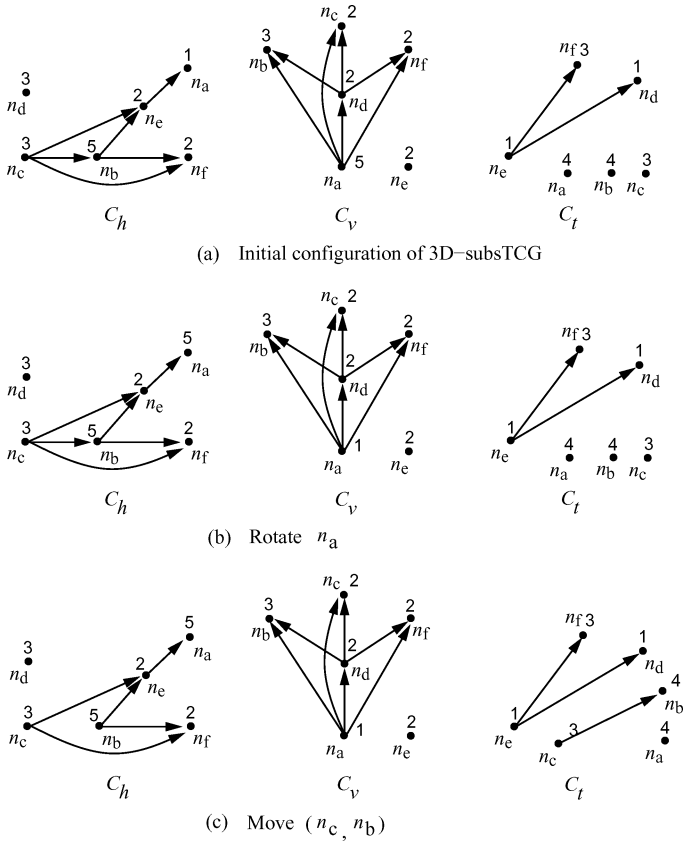

Fig. 13. Examples of perturbations. (a) The initial 3D-subTCG (C_h, C_v, and C_t); (b) the resulting 3D-subTCG after rotating the task n_a shown in (a); (c) the resulting 3D-subTCG after moving the reduction edge (n_c, n_b) from the C_h of (b) to C_t.

Note that Rotation, Swap, Reverse, and Move are first introduced in Lin and Chang [2001], which can be performed in respective $O(1)$, $O(1)$, $O(m^2)$, and $O(m^2)$ time, where m is the number of tasks. The Rotation and Swap operations do not change the topology of 3D-subTCG, while Reverse, Move, and Transpositional Move do. Therefore, to maintain the properties of a 3D-subTCG, we may need to update the resulting graphs after performing Reverse, Move, and Transpositional Move. Furthermore, we shall perform feasibility detection during each perturbations to ensure that the precedence constraints are not violated. We first detail the operations as follows.

4.2.1 *Rotation.* To rotate a *task* v_i, we only need to exchange the weights of the corresponding node n_i in C_h, C_v, and C_t. Figure 13(b) shows the resulting 3D-subTCG after rotating the task v_a in Figure 13.

4.2.2 *Move.* The Move operation moves a *reduction* edge (n_i, n_j) in one graph to one of the other two graphs in a 3D-subTCG. Move could switch the relations of the two tasks v_i and v_j between a horizontal relation and a vertical

one. For two tasks v_i and v_j, $v_i \vdash v_j$ ($v_i \perp v_j$) and there exists a reduction edge (n_i, n_j) in C_h (C_v); after moving the edge (n_i, n_j) to C_v (C_h), we have the new geometric relation $v_i \perp v_j$ ($v_i \vdash v_j$). Move could also change the temporal relation of the two tasks v_i and v_j. For two tasks v_i and v_j, $v_i \prec v_j$ and there exists a reduction edge (n_i, n_j) in C_t; after moving the edge (n_i, n_j) to C_h (C_v), we change the temporal relation into the new geometric relation $v_i \vdash v_j$ ($v_i \perp v_j$). If there exists a reduction edge (n_i, n_j) in C_h (C_v); after moving the edge (n_i, n_j) to C_t, we have the new temporal relation $v_i \prec v_j$.

To move a reduction edge (n_i, n_j) from one transitive graph G to another transitive graph G', we first delete the edge (n_i, n_j) from G and then add (n_i, n_j) to G'. For each node $n_k \in F_{in}(n_i) \cup \{n_i\}$ and $n_l \in F_{out}(n_j) \cup \{n_j\}$, we shall check whether the edge (n_k, n_l) exists in G'. If G' contains this edge, we do nothing; otherwise, we need to add this edge to G' and delete the corresponding edge (n_k, n_l) or (n_l, n_k) in G or G'', if any, to maintain the properties of the 3D-subTCG. Figure 13(c) shows the result of moving the edge (n_c, n_b) in C_h of Figure 13(b) to C_t.

4.2.3 *Swap.* To swap the nodes n_i and n_j of two tasks v_i and v_j, we only need to exchange the nodes n_i and n_j in C_h, C_v, and C_t. Figure 14(a) shows the result of swapping nodes n_b and n_d shown in Figure 13(c).

4.2.4 *Reverse.* The Reverse operation reverses the direction of a *reduction* edge (n_i, n_j) in one graph. For two modules v_i and v_j, $v_i \vdash v_j$ ($v_i \perp v_j$) and there exists a reduction edge (n_i, n_j) in C_h (C_v); after reversing the edge (n_i, n_j), we have the new geometric relation $v_j \vdash v_i$ ($v_j \perp v_i$). Similarly, $v_i \prec v_j$ and there exists a reduction edge (n_i, n_j) in C_t; after reversing the edge (n_i, n_j), we have the new temporal relation $v_j \prec v_i$.

To reverse a reduction edge (n_i, n_j) in a transitive graph G, we first delete the edge from G, and then add the edge (n_j, n_i) into G. Similar to the Move operation, for each node $n_k \in F_{in}(n_j) \cup \{n_j\}$ and $n_l \in F_{out}(n_i) \cup \{n_i\}$ in G, we shall check whether the edge (n_k, n_l) exists in G. If G contains the edge, we do nothing; otherwise, we need to add the edge to G and delete the corresponding edge (n_k, n_l) or (n_l, n_k) in the other two transitive closure graphs, if any, to maintain the properties of the 3D-subTCG. Figure 14(b) shows the result after reversing the edge (n_e, n_a) in C_h of Figure 14(a).

4.2.5 *Transpositional Move.* The Transpositional Move operation removes a *reduction* edge (n_i, n_j) from one graph, and adds an edge (n_j, n_i) to one of the two graphs in a 3D-subTCG. In one case, Transpositional Move switches the geometric relation of the two tasks v_i and v_j between a horizontal relation and a vertical one and changes the ordering of the two tasks v_i and v_j in their geometric relation. For two tasks v_i and v_j, $v_i \vdash v_j$ ($v_i \perp v_j$) and there exists a reduction edge (n_i, n_j) in C_h (C_v); after transpositionally moving the edge (n_i, n_j) to C_v (C_h), we have the new geometric relation $v_j \perp v_i$ ($v_j \vdash v_i$). In the other case, Transpositional Move changes the temporal relation of the two tasks v_i and v_j. For two tasks v_i and v_j, $v_i \prec v_j$ and there exists a reduction edge (n_i, n_j) in C_t; after transpositionally moving the edge (n_i, n_j) to C_h (C_v), we change the temporal relation into the new geometric relation $v_j \vdash v_i$ ($v_j \perp v_i$). If there

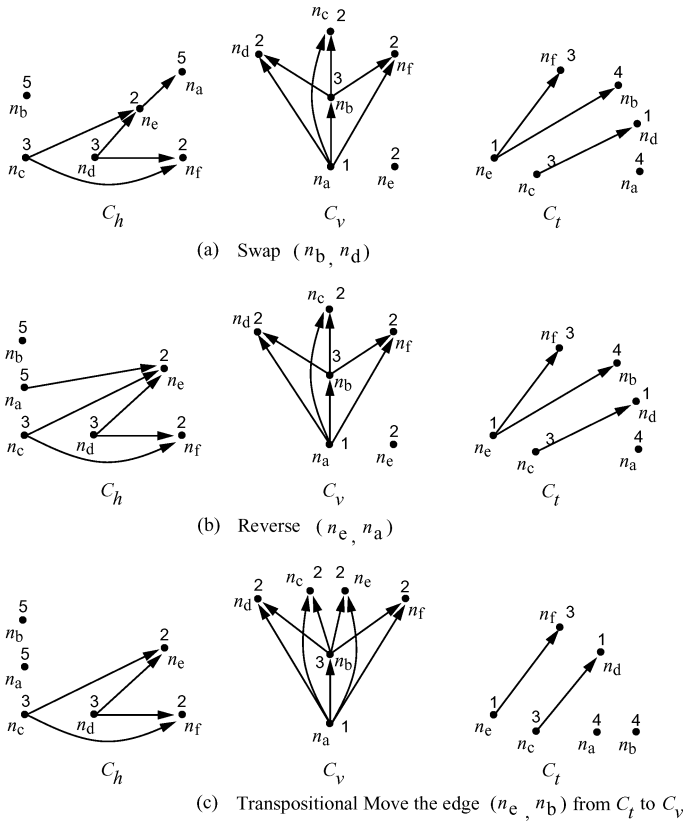

Fig. 14. Examples of perturbations (continued from Figure 13). (a) The resulting 3D-subTCG after swapping the nodes n_b and n_d shown in Figure 13 (c); (b) the resulting 3D-subTCG after reversing the reduction edge (n_e, n_a) in the C_h shown in (a); (c) the resulting 3D-subTCG after transpositional moving the reduction edge (n_e, n_b) from the C_t of (b) to C_v.

exists a reduction edge (n_i, n_j) in C_h (C_v); after transpositionally moving the edge (n_i, n_j) to C_t, we have the new temporal relation $v_j \prec v_i$.

To transpositionally move a reduction edge (n_i, n_j) from one transitive graph G to another transitive graph G', we first delete the edge (n_i, n_j) from G and add (n_j, n_i) to G'. Similar to the Move operation, for each node $n_k \in F_{in}(n_j) \cup \{n_j\}$ and $n_l \in F_{out}(n_i) \cup \{n_i\}$, we shall check whether the edge (n_k, n_l) exists in G'. If G' contains the edge, we do nothing; otherwise, we need to add the edge to G' and delete the corresponding edge (n_k, n_l) or (n_l, n_k) in G or G'', if any, to maintain the properties of the 3D-subTCG. Figure 14(c) shows the result of transpositionally moving the edge (n_e, n_b) from C_t of Figure 14(b) to C_v. Note we delete the edge (n_a, n_e) in C_h and add it to C_v. Figure 15 shows the resulting placement of 3D-subTCG shown in Figure 14(c).

Note that the Transpositional Move operation is different from performing Move followed by Reverse. Figure 16 shows the resulting 3D-subTCG if we first Move the edge (n_e, n_b) from C_t to C_v and then Reverse the edge (n_e, n_b) in C_v.

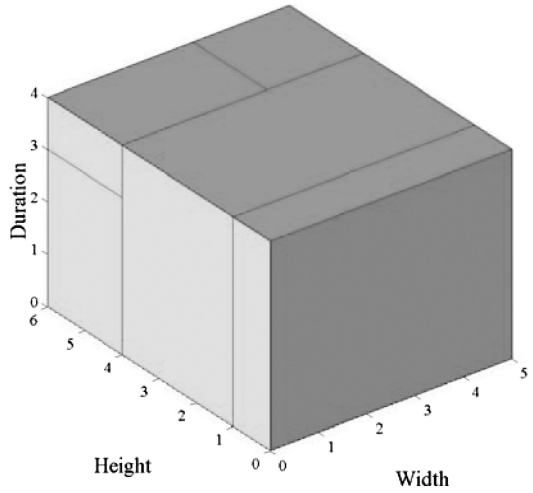

Fig. 15. The resulting placement of 3D-subTCG shown in Figure 14(c).

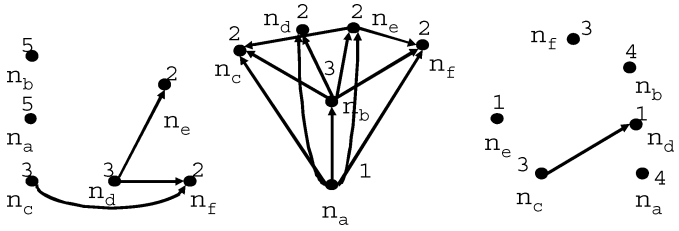

Fig. 16. The resulting 3D-subTCG after Move the edge (n_e, n_b) from C_t to C_v then Reverse the edge (n_e, n_b) in C_v.

It is clear that this 3D-subTCG is not the same as the 3D-subTCG shown in Figure 14(c).

After updating the resulting 3D-subTCG after Move, Reverse, Transpositional Move, we preserve the three properties of 3D-subTCG. However, due to the existence of two pathes between two nodes in different graphs, the Move and the Transpositional Move may potentially generate a 3D-subTCG with cycles. Figure 17 shows this scenario. Figure 17(a) shows a feasible 3D-subTCG with five nodes. If we Move the edge (n_d, n_e) from C_t to C_h, we need to add edges (n_d, n_a), (n_d, n_b), and (n_d, n_c) into C_h in order to maintain the properties of 3D-subTCG. However, since we delete edges (n_c, n_d) and (n_d, n_a) in C_v, the edge (n_c, n_a) becomes a reduction edge. If we Move edge (n_c, n_a) to C_h, then a cycle between nodes n_a, n_b, and n_c is generated as shown in Figure 17(c). In order to avoid this situation, we do not Move or Transitional Move a reduction edge that will generate cycles.

4.3 Feasibility Detection

To maintain the temporal ordering among tasks, the 3D-subTCG must guarantee that all precedence constraints are satisfied. Among the five operations

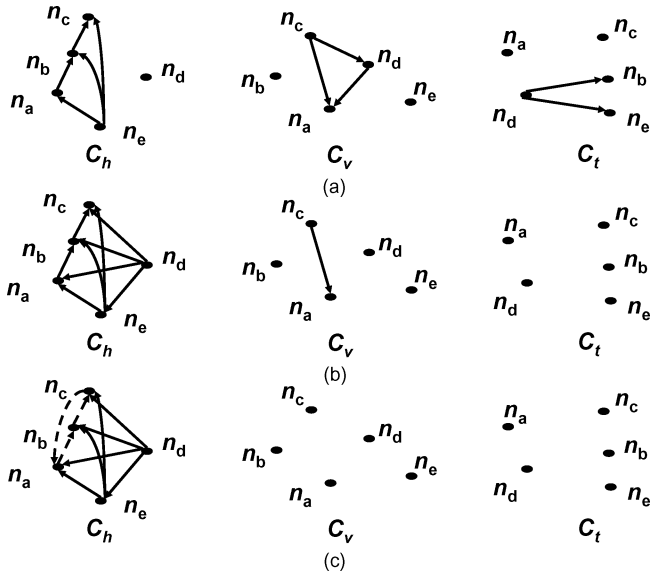

Fig. 17. (a) A feasible 3D-subTCG; (b) the resulting 3D-subTCG if we Move edge (n_d, n_e) from C_t to C_h; (c) if we Move the reduction edge (n_c, n_a) from C_v to C_h, then a cycle consisted of edges (n_a, n_b), (n_b, n_c), and (n_c, n_a) (shown in dash lines) is generated.

mentioned above, Move, Swap, Reverse, and Transpositional Move could violate the constraints. We now show how to detect a violation during perturbation.

When we move an edge (n_i, n_j) or reverse/transpositionally move (n_j, n_i), the precedence constraint will be violated if $n_l \in F_{in}(n_i) \cup \{n_i\}$, $n_k \in F_{out}(n_j) \cup \{n_j\}$, and $(n_l, n_k) \notin C_t$ if $(n_l, n_k) \in D$. As mentioned in Section 2, D denotes the precedence constraints. When we swap two nodes n_i and n_j, three scenarios could happen:

(1) There exists a precedence constraint between n_i and n_j,
(2) Neither of n_i and n_j has a precedence constraint, or
(3) Either n_i or n_j has precedence constraint.

In the first case, it is clear that we cannot swap the two nodes. However, if neither of n_i and n_j has a precedence constraint, we can swap n_i and n_j directly. Without loss of generality, we could assume that node n_i has precedence constraints to explain the third case. If n_i has a precedence-constrained edge (n_i, n_k), we can swap n_i and n_j without any violation if $n_k \in F_{out}(n_j)$ of C_t. If n_i has a precedence-constrained edge (n_k, n_i) and $n_k \in F_{in}(n_j)$ of C_t, we can also swap n_i and n_j.

Figure 18(a) shows the resulting C_h, C_v, and C_t after swapping the nodes n_d and n_e in Figure 14(c). Assume that there exists a precedence-constrained edge (n_e, n_f). The precedence constraint will be violated if we swap the two nodes n_d and n_e, since $n_f \notin F_{out}(n_d)$ of C_t. Figure 18(b) shows C_h, C_v, and C_t after reversing the edge (n_d, n_e) in the C_h in Figure 14(c). Since $\{n_e\} \cap F_{in}(n_e) = \{n_c, n_e\}$

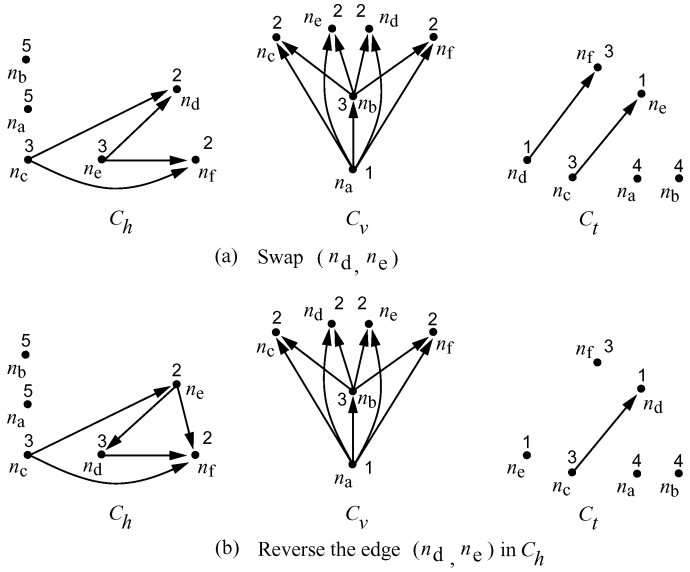

Fig. 18. (a) The resulting 3D-subTCG after swapping the nodes n_d and n_e shown in Figure 14(c); (b) the resulting 3D-subTCG after reversing the reduction edge (n_d, n_e) in the C_h shown in Figure 14(c).

and $\{n_d\} \cap F_{out}(n_d)=\{n_d, n_f\}$ in C_h, we shall check (n_e, n_f) for the precedence constraint. If there exists a precedence-constrained edge (n_e, n_f), the precedence constraint will be violated.

By doing the feasibility detection during the Move, Reverse, Transpositional Move, and Swap operations, we can guarantee that the resulting 3D-subTCG still satisfies all precedence constraints. We thus have the following theorem.

THEOREM 4.1. *The precedence constraints of a 3D-subTCG are not violated by the Move, Swap, Reverse, or Transpositional Move operation with the feasibility detection.*

PROOF.

(1) Recall that if an edge (n_i, n_j) is in C_t, then task v_j must be executed after task v_i. Thus, the necessary condition to satisfy the precedence constraint between v_i and v_j is that the edge (n_i, n_j) must be in C_t.
(2) If an edge (n_i, n_j) is a precedence-constrained edge, we will not choose this edge to perform Move, Reverse, or Transitional Move. We will not swap nodes n_i and n_j, either.
(3) When we Move, Reverse, or Transitional Move an edge (n_i, n_j), if a precedence-constrained edge (n_l, n_k) should be deleted from C_t and added to another graph, we abort this operation and choose another reduction edge. It guarantees that all the precedence-constrained edges remain in C_t.
(4) If there exists a precedence-constrained edge (n_i, n_j) and we want to swap node n_i with n_k or n_j with n_k, there must exist an edge (n_k, n_j) or edge

(n_i, n_k). This guarantees that the precedence-constrained edge (n_i, n_j) still exists after performing Swap.

Thus, by doing the feasibility detection, the precedence constraints are not violated by the Move, Reverse, Transitional Move, and Swap. □

5. TEMPORAL FLOORPLANNING WITH CONSTRAINTS

In this section, we extend our temporal floorplanning algorithm to handle placement with boundary modules and the fixed-outline constraint.

5.1 Boundary Constraint

The floorplanning with boundary constraint is to place a set of tasks on the desired boundary of a reconfigurable device and can be formulated as follows:

Definition 5.1. *Boundary constraint.* Given a boundary module v_i, it must be placed on one of the four sides: on the left, on the right, at the bottom, or at the top of a reconfigurable device in the final packing.

Since the geometric relation of each pair of tasks is transparent to the 3D-subTCG representation, we can easily detect if a task is on the boundary of a reconfigurable device. Given a task v_i, we can detect if v_i is in one of the four boundaries of a reconfigurable device by observing the in-degree/out-degree of the corresponding node n_i in C_h or C_v. If the in-degree of n_i in C_h (C_v) is zero, then v_i is placed on the left (bottom) boundary of a reconfigurable device. Similarly, if the out-degree of n_i in C_h (C_v) is zero, then v_i is placed on the right (top) boundary of a reconfigurable device.

In this paper, for each boundary module v_i, we associate it with a *boundary penalty*, BP_i, and is defined as follows:

$$BP_i = \begin{cases} 0, & if\ v_i\ is\ on\ the\ boundary \\ d(n_i), & if\ v_i\ is\ not\ on\ the\ boundary, \end{cases} \tag{3}$$

where $d(n_i)$ is the distance of n_i to the source node in C_h (C_v) if v_i needs to be placed on the left (bottom) boundary or the distance of n_i to the sink node in C_h (C_v) if v_i needs to be placed on the right (top) boundary. We can use the above method to detect whether v_i is on the boundary of a reconfigurable device. We add the boundary penalty in the cost function. The modified cost function ϕ' is given by:

$$\Phi' = \alpha(1 + B/B_{norm})V + \beta W + \gamma O, \tag{4}$$

where B is the summation of BP_i for all boundary modules v_i and B_{norm} is used for normalization. B is zero if all BP_is are equal to zero.

5.2 Fixed-Outline Floorplanning

In this subsection, we handle the fixed-outline constraint imposed by a given reconfigurable device. 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 floorplan of

fixed-outline floorplanning must satisfy the outline constraint; that is, $W_p \leq W_f$ and $H_p \leq H_f$. In this paper, we add the excessive volume in the cost function to ensure the satisfaction of the fixed-outline constraint. The modified cost function Φ'' is given by:

$$\Phi'' = \alpha V' + \beta W + \gamma O + \delta M, \tag{5}$$

where V' is defined as $W_f \times H_f \times Time$, δ is a user-specified constant, and M is the penalty term and is given by the following equation:

$$\begin{aligned} M \;=\; & \min((\max(W_p - W_f, 0) \times H_p) \times Time \\ & +(\max(H_p - H_f, 0) \times W_p) \times Time, \\ & (\max(H_p - W_f, 0) \times W_p) \times Time \\ & +(\max(W_p - H_f, 0) \times H_p) \times Time). \end{aligned} \tag{6}$$

Since the whole design can be rotated by 90 degrees, we choose the smaller excessive volume of two orthogonal placements. The rationale behind M is that when simulated annealing minimizes the cost function, it automatically minimizes the penalty term. Thus, the fixed-outline constraint is automatically satisfied.

Beside considering the excessive volume in the objective function, we determine the probability of the Move and Transpositional Move operations for moving a reduction edge from C_h or C_v to C_t based on the value p defined in the following equation:

$$p = \max\left(\frac{Q \times k}{Q \times k + 1}, 0\right), \tag{7}$$

where k is a user-specified value and Q is defined by

$$Q = \max\left(\frac{W_p - W_f}{W_f}, \frac{H_p - H_f}{H_f}\right). \tag{8}$$

In this paper, we set k equal to 1.25. At each perturbation, we calculate p based on the width and height of the current placement. The larger value of p, the harder for a floorplan to fit the desired outline. Therefore, we should increase the probability to Move or Transpositional Move a reduction edge from C_h or C_v to C_t to satisfy the fixed-outline constraint for a larger p.

6. EXPERIMENTAL RESULTS

Based on simulated annealing [Kirkpatrick et al. 1983], we implemented the temporal floorplanning algorithm in the C++ programming language on a 433 MHz SUN Ultra-60 workstation with 1 GB memory. We compared 3D-subTCG with Sequence Triplet (ST) [Yamazaki et al. 2000] and T-tree based on the same SA engine and same SA parameters, (cooling schedule, initial temperature, weights of the cost function, etc.). ST is extended from the well-known Sequence Pair (SP) [Murata et al. 1995] representation, which is very popular for handling floorplanning/placement in both the industry and academia.

In this section, we first report the outline-free floorplanning results. Then, we report results for boundary constraint and fixed-outline constraint.

Table II. Results of Volume Optimization (volume = $mm^2\times$ clock cycles)

Circuit	# of tasks	Sum of volume	Volume	Dead space (%)	time (Sec.)
Circuit 1	10	512	512	0.0	8.3
Circuit 2	10	480	480	0.0	1.9
Circuit 3	10	1000	1000	0.0	9.7
Circuit 4	20	3840	4032	4.7	25.2
Circuit 5	30	4096	4608	11.1	127.8

Table III. Results for Volume and Overhead Optimization

			ST			3D-subTCG		
Circuit	# of tasks	Sum of volume	Volume ($mm^2\times$ clks)	Dead Space (%)	Time (sec.)	Volume ($mm^2\times$ clks)	Dead Space (%)	Time (sec.)
beasley1	10	6218	8710	28.6	7.7	7504	17.1	8.5
beasley2	17	11497	14664	21.5	45.2	12402	7.2	28.5
beasley3	21	10362	16016	35.3	44.1	12640	18.0	22.4
beasley4	7	10205	13800	26.0	3.0	13064	21.8	2.0
beasley5	14	16734	22750	26.4	18.2	18912	11.5	16.0
beasley6	15	11040	14994	26.3	27.9	13200	16.3	24.8
beasley7	8	17168	24570	30.1	3.8	20574	16.5	2.3
beasley8	13	83044	132275	37.2	15.4	98280	15.5	19.4
beasley9	18	133204	174496	23.6	30.6	167751	20.5	17.2
beasley10	13	493746	660480	25.2	13.0	575685	14.2	10.8
beasley11	15	383391	486381	24.8	17.5	438702	12.6	9.8
beasley12	22	646158	922080	29.9	100.0	823816	21.5	58.5
okp1	50	1.24327e+08	216950048	42.6	1607.2	173829024	28.4	387.3
okp2	30	8.54452e+07	128093128	33.2	285.3	110095000	22.3	73.8
okp3	30	1.23808e+08	185146208	33.1	280.7	160854400	23.0	70.6
okp4	61	2.38861e+08	417942304	42.8	791.3	328835424	27.3	501.9
okp5	97	1.89875e+08	448984000	57.7	607.8	295849984	35.8	565.9
Average				32.01			19.38	

6.1 Results for Outline-Free Floorplanning

In this subsection, we report the result for outline-free floorplanning problem. We have conducted four sets of experiments: (1) volume optimization, (2) volume and overhead optimization, (3) simultaneous volume, wirelength, and overheads optimization, and (4) volume and overhead optimization for five real circuits.

To verify our algorithm, we first tested 3D-subTCG on five synthetic circuits that can be packed without deadspace. Table II shows the results. Note that the volume of a placement is the minimum bounding box enclosing the placement. We can see that 3D-subTCG obtains the optimal placements for the first three test cases and near optimal solutions for the last two larger circuits, all in reasonable time. The results show that our approach is very effective for cost optimization.

For the second experiment, we perform volume and reconfiguration and communication overheads optimization. In this experiment, we adopted the benchmark circuits used in Fekete and Schepers [1997] and added the reconfiguration and communication overheads. We compared 3D-subTCG with ST. As shown in Table III, the 3D-subTCG based method outperforms the ST-based one by a

Table IV. The Five 3D-MCNC Benchmark Circuits

Circuit	# of modules	# of pads	# of nets	# pins	Total volume	# of precedence constraints
3D-apte	9	73	97	214	9.8×10^7	3
3D-xerox	10	107	203	696	4.0×10^7	3
3D-hp	11	43	83	264	1.2×10^7	3
3D-ami33	33	42	123	480	2.3×10^6	7
3D-ami49	49	24	408	931	1.3×10^8	11

large margin. 3D-subTCG achieved less deadspace on the average compared to ST (19.38% vs. 32.01%).

For the third experiment, we perform 3D placement with the considerations of precedence constraints, wirelength, and reconfiguration/communication overheads. In this experiment, we used the MCNC benchmarks. Since the MCNC benchmarks do not have execution time and precedence constraints, we assigned their execution time and precedence constraints by ourselves. The new benchmark suite is called the 3D-MCNC benchmark. Table IV lists the statistics of the five 3D-MCNC benchmarks. In the following, we describe how to construct the control data flow graph (CDFG) from a traditional floorplanning benchmark. The basic idea is to construct the edges among tasks based on the interconnections among them.

Let F stands for the given floorplanning benchmark. First, we define c_i for each task v_i as the summation of the number of interconnections between v_i and all other tasks plus the number of tasks that are connected to v_i. Let C be the set of c_i sorted in descending order. The initial CDFG $DG = \{S, E\}$ contains the first k tasks in C, where k is a user-defined constant. Then we delete the interconnections among these k tasks in F. We iteratively add tasks and edges into DG. For each iteration, we add at most r directed edges into E from H, where $H = \{(v_i, v_j) | v_i \in S, v_j \notin S$, and there exist interconnections between v_i and v_j in $F\}$. Here, r is a user-specified constant. Then we add the task v_j into DG and delete the interconnections associated with the task v_j and the interconnections between the tasks v_i and v_j. When all tasks are in DG, the algorithm terminates. Finally, we randomly select l tasks from DG and the edges in E that connect these tasks to form the final CDFG. Here, l is a user-specified constant.

In this experiment, we simultaneously optimized volume and wirelength, and reconfiguration/communication overheads with precedence constraints. We compared 3D-subTCG with ST and T-tree. Table V shows the results. As shown in Table V, 3D-subTCG achieves better volume utilization (15% deadspace v.s. 35% deadspace) and shorter wirelength compared with ST. 3D-subTCG also needs less CPU time than ST. Compared with T-tree, 3D-subTCG obtains comparable deadspace (14.86% v.s. 14.22%) with shorter wirelength (396.08 v.s. 402.06). Figure 19 shows the resulting placement of 3D-xerox.

Although it is hard to quantify, a key insight to the different performance between 3D-subTCG and (ST) lies in the effects of their perturbations: swapping two modules in a ST may lead to a dramatic change from the original placement while the change for the 3D-subTCG perturbation is smaller, which makes simulated annealing easier to converge to an optimal solution. (Here is

Table V. Results of Volume and Wirelength Optimization for the Five 3D-MCNC Benchmark Circuits

	ST				3D-subTCG			
Circuit	Volume ($mm^2\times$ clock cycles)	Wire-length (mm)	Dead space (%)	Time (sec.)	Volume ($mm^2\times$ clock cycles)	Wire-length (mm)	Dead space (%)	Time (sec.)
3D-apte	1.1×10^8	495.0	16.2	7.7	1.0×10^8	335,3	5.9	3.9
3D-xerox	5.2×10^7	613.2	23.1	19.5	4.4×10^7	602.0	8.4	8.9
3D-hp	2.0×10^7	387.3	37.2	20.6	1.5×10^7	158.3	13.7	11.2
3D-ami33	4.1×10^6	84.7	44.5	446.4	3.0×10^6	77.7	24.7	128.1
3D-ami49	2.9×10^8	1040.8	54.9	1066.7	1.6×10^8	807.1	21.6	680.2
Average		524.2	52.9	312.18		396.08	14.86	166.46
	T-tree				3D-subTCG			
3D-apte	1.0×10^8	380.0	5.9	0.58	1.0×10^8	335.3	5.9	3.9
3D-xerox	4.7×10^7	595.7	13.8	1.78	4.4×10^7	602.0	8.4	8.9
3D-hp	1.41×10^7	165.1	8.6	1.65	1.5×10^7	158.3	13.7	11.2
3D-ami33	3.0×10^6	78.1	24.5	34.33	3.0×10^6	77.7	24.7	128.1
3D-ami49	1.61×10^8	791.4	18.3	72.46	1.6×10^8	807.1	21.6	680.2
Average		402.06	14.22	22.16		396.08	14.86	166.46

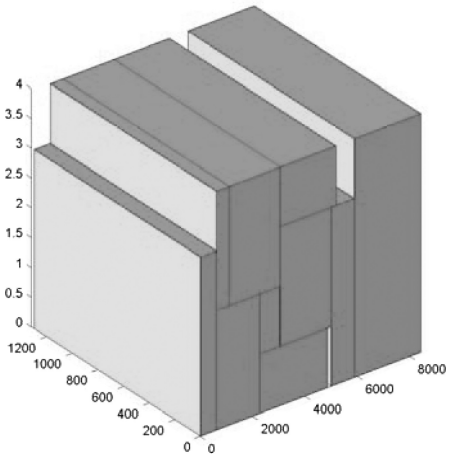

Fig. 19. The result of 3D-xerox with optimizing volume and wirelength simultaneous.

an analogy: like the gradient search for the optimization of nonlinear programming, the step size plays an important role in determining whether a search scheme can converge to the global optimal solution—a huge step size may fail to converge to an optimal solution.)

For the final experiment, we used five real circuits: JPEG encoder [Banerjee et al. 2005], 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 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 VI shows the result of the five real circuits. Columns 2 and 3 list the number of tasks and the number of precedence constraints for each circuit, respectively. Column 4 gives the total volume of each circuit.

Table VI. Results of Volume and Overhead Optimization for Five Real Circuits

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

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 3D-subTCG obtains comparable average volumes (24.52% deadspace vs. 24.64% deadspace) and needs longer average CPU time (116.9 sec. vs. 15.54 sec.) than T-tree. This experiment demonstrates the ability of 3D-subTCG to handle the real circuits. It also confirms our observation in Section 3.3 that T-tree has advantages in packing efficiency and volume optimization, especially for large-scale circuits, such as the FFT circuit.

6.2 Results for Boundary Constraints and Fixed-Outline Constraints

In this subsection, we first report the result for boundary constraint. Next, we report the result for fixed-outline constraint.

For the floorplanning with boundary modules, we compared 3D-subTCG with T-tree. The goal of this experiment is to verify the ability of 3D-subTCG for handling various floorplanning constraints. For T-tree, we discarded the infeasible solutions (boundary modules are not on the boundary) during simulated annealing. We used the 3D-MCNC benchmarks and the five real circuits for this experiment. Tables VII and VIII show the respective results for the 3D-MCNC benchmarks and the five real circuits. For the 3D-MCNC benchmark, we considered the volume, wirelength, and overhead, while we considered the volume and overhead for the five real circuits. In both tables, column 2 shows the numbers of the top, bottom, left, and right tasks, denoted by $\#|T|$, $\#|B|$, $\#|L|$, and $\#|R|$, respectively. As shown in Table VII, 3D-subTCG achieves shorter average wirelength (358.94 mm vs. 388.88 mm) and smaller average deadspace (17.84% vs. 18.3%) than T-tree for the 3D-MCNC benchmarks. However, 3D-subTCG needs longer CPU time (99.18 sec vs. 27.81 sec) than T-tree. Similar results are also obtained for the five real circuits. For the five real circuits, 3D-subTCG obtains smaller average volume (24.09% deadspace vs. 26.00% deadspace) with longer CPU time (113.28 sec vs. 47.15 sec) than T-tree. From Tables VI and VIII, we observe that 3D-subTCG obtains similar volumes as T-tree if no boundary constraint is considered, and obtains smaller volumes than T-tree if boundary constraints need to be addressed. The experimental results confirm our observation described in Section 3.3 that 3D-subTCG may be more suitable for handling various floorplanning constraints, because 3D-subTCG keeps more geometric information in the representation and has a larger solution space than T-tree. We can easily determine if a task is on the boundary of a device

Table VII. Results for the 3D-MCNC Benchmarks with Boundary Constraints

		T-tree			
Circuit	#|T|, #|B|, #|L|, #|R|	Volume ($mm^2\times$ clock cycles)	Wire-length (mm)	Dead Space (%)	Time (sec.)
3D-apte	1,1,1,1	1.05×10^8	341.0	5.9	2.25
3D-xerox	1,1,1,1	4.67×10^7	400.5	13.2	2.02
3D-hp	1,1,1,1	1.63×10^7	201.1	20.9	2.82
3D-ami33	2,2,2,2	3.19×10^6	61.5	27.2	40.13
3D-ami49	3,3,2,3	1.74×10^8	940.3	24.3	91.86
average			388.88	18.3	27.81
		3D-subTCG			
Circuit	#|T|, #|B|, #|L|, #|R|	Volume ($mm^2\times$ clock cycles)	Wire-length (mm)	Dead Space (%)	Time (sec.)
3D-apte	1,1,1,1	1.05×10^8	311.0	5.9	2.68
3D-xerox	1,1,1,1	4.66×10^7	447.6	13.2	7.99
3D-hp	1,1,1,1	1.50×10^7	176.0	13.7	1.67
3D-ami33	2,2,2,2	3.31×10^6	67.4	30.0	115.6
3D-ami49	3,3,2,3	1.79×10^8	792.7	26.4	368.0
average			358.94	17.84	99.18

Table VIII. Results for the Five Real Circuits with Boundary Constraints

		T-tree		
Circuit	#|T|, #|B|, #|L|, #|R|	Volume	Dead Space (%)	Time (sec.)
JPEG encoder	0,1,1,1	25785	31.0	0.55
RLS	1,1,1,1	25500	27.7	1.06
FIR	1,1,2,1	46440	8.1	5.33
BF	2,2,2,1	46696	25.8	6.06
FFT	3,3,3,3	153180	37.4	222.76
average			26	47.15
		3D-subTCG		
Circuit	#|T|, #|B|, #|L|, #|R|	Volume	Dead Space (%)	Time (sec.)
JPEG encoder	0,1,1,1	25785	31.0	5.03
RLS	1,1,1,1	24150	23.6	13.19
FIR	1,1,2,1	45824	6.8	6.97
BF	2,2,2,1	44850	22.7	24.6
FFT	3,3,3,3	150696	36.38	516.63
average			24.09	113.28

by checking the indegree and outdegree of its corresponding node in C_h or C_v, and thus the SA engine can search for the feasible solutions more effectively. Figure 20 shows the resulting 3D floorplan of 3D-ami49. White modules represent boundary modules.

Table IX. Results for Various Aspect Ratios of Desired Widths and Heights for 3D-ami33 Circuit

Circuit name	Outline width/ height	Outline-free SA engine: Success rate	Min/Avg/Max Exec. time (clk cycles)	Min/Avg/Max Deadspace (%)
3D-ami33	1100/600	47%	6/6.61/11	21.51/29.87/39.11
	900/900	13%	6/7.92/11	23.81/32.72/37.66
	850/700	9%	7/8.44/11	24.47/34.30/37.66
	550/1200	42%	6/6.47/11	21.51/29.98/40.02
	650/800	6%	7/8.66/11	24.47/33.79/37.44
Avg.		23.4%	6.4/7.62/11	23.15/32.13/38.37
Circuit name	**Outline width/ height**	**Fixed-outline SA engine: Success rate**	**Min/Avg/Max Exec. time (clk cycles)**	**Min/Avg/Max Deadspace (%)**
3D-ami33	1100/600	91%	6/6.87/8	37.85/46.15/55.60
	900/900	92%	6/6.66/8	46.45/54.56/60.77
	850/700	69%	6/7.62/10	32.70/45.89/59.36
	550/1200	58%	6/7.56/10	34.17/49.58/64.46
	650/800	46%	7/8.59/11	31.73/45.11/57.72
Avg.		71.2%	6.2/7.46/9.4	36.58/48.25/59.58

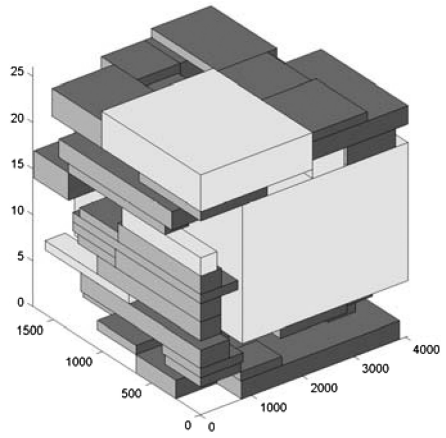

Fig. 20. The result of 3D-ami49 with boundary constraints. White modules represent boundary modules.

For the fixed-outline floorplanning problem, we chose the 3D-ami33 circuit for experiment. We added various outline constraints. Table IX reports the success rate[2], the minimum/average/maximum task execution time[3] and the minimum/average/maximum deadspace[4] of the fixed-outline SA engine described in Section 5.2. We follow, Adya and Markov [2001, 2003] to compare the success rate with and without considering the fixed-outline constraint. The

[2]Number of runs that satisfies the fixed-outline constraint in 100 runs.
[3]The minimum/average/maximum total execution time in all successful runs.
[4]The minimum/average/maximum deadspace in all successful runs.

minimum/average/maximum execution time (clk) listed in this table is the total execution time of the scheduled tasks. The minimum/average/maximum deadspace is the deadspace in all successful runs. In this experiment, we set different ratios of desired widths and heights for 3D-ami33. 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 obtain higher success rates (71.2% vs. 23.4%) with smaller minimum total execution time (6.2 vs. 6.4) and larger minimum deadspace (36.58% vs. 23.15%) compared with outline-free SA engine. The result shows the effectiveness of our fixed-outline SA engine. One observation is that for the 850/700 outline constraint, outline-free SA engine obtains larger minimum execution time with smaller minimum deadspace than fixed-outline SA engine. The reason is that fixed-outline SA engine makes use of the given architecture, and therefore may generate a floorplan with smaller execution time. In contrast, free-outline SA engine optimizes volume, and therefore may generate a floorplan with longer execution time but smaller area, hence smaller deadspace.

7. CONCLUSIONS AND DISCUSSIONS

We have presented the 3D-subTCG representation to handle the temporal floorplanning/placement problem for dynamically reconfigurable FPGAs. We have explored the feasibility conditions for the temporal relations among tasks/modules. Our algorithm can guarantee a feasible placement in each perturbation. Experimental results have shown that our method is very effective and efficient for temporal floorplanning/placement.

Luk at al. [1997] discussed the idea that we can reduce the reconfiguration overhead by considering the similarities between tasks. Recently, Ghiasi and Sarrafzadeh [2003] proposed an optimal algorithm to reduce the huge reconfiguration overhead by exploiting the similarity among tasks. Our future work will consider these factors when evaluating the reconfiguration overhead. There are two possible ideas. The first one is that if the configuration of each cell for each task is known, we can calculate how many reconfiguration bits are needed to configure a task. Therefore, we can obtain more accurate reconfiguration overhead between two tasks. The other possible idea is that we can group several tasks with similar functionality, since their configurations may be the same.

REFERENCES

ADYA, S. N. AND MARKOV, I. L. 2001. Fixed-outline floorplanning through better local search. In *Proceedings of IEEE International Conference of Computer Design*. 328–334.

ADYA, S. N. AND MARKOV, I. L. 2003. Fixed-outline floorplanning: Enabling hierarchical design. *IEEE Trans Very Large Scale Integra. Syst. 11*, 6 (Dec.), 1120–1135.

ALSOLAIM, A., BECKER, J., GLESNER, M., AND STARZYK, J. 2000. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 205–214.

ATMEL. 1997. *AT6000 FPGA Configuration Guide Documentation 0436B*. Atmel, Inc.

BANERJEE, S., BOZORGZADEH, E., AND DUTT, N. 2005. Hw-sw partitioning for architectures with partial dynamic reconfiguration. Tech. rep. CECS-TR-05-02, UC Irvine.

BAZARGAN, K., KASTNER, R., AND SARRAFZADEH, M. 2000a. 3-d floorplanning: Simulated annealing and greedy placement methods for reconfigurable computing systems. *Design Autom. for Embed. Syst.*—RSP'99 Special Issue.

BAZARGAN, K., KASTNER, R., AND SARRAFZADEH, M. 2000b. Fast template placement for reconfigurable computing systems. *IEEE Design Test Comput. 17*, 1 (March), 68–83.

BAZARGAN, K. AND SARRAFZADEH, M. 1999. Fast online placement for reconfigurable computing systems. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 300–302.

CHAUBAL, A. P. 2004. Design and implementation of an FPGA-based partially reconfigurable network controller. Master thesis, Virginia Polytechnic Institute and State University.

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

DURBANO, J. P. AND ORTIZ, F. E. 2004. FPGA-based acceleration of the 3d finite-difference time-domain method. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 156–163.

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

FEKETE, S. P. AND SCHEPERS, J. 1997. On more-dimensional packing iii: Exact algorithms. ZPR Tech. Rep. 97-290.

GHIASI, S. AND SARRAFZADEH, M. 2003. Optimal reconfiguration sequence management. In *Proceedings of Asia-South Pacific Design Automation Conference*. 359–365.

HAUCK, S. 1998. The roles of FPGAs in reprogrammable systems. *Proceedings of IEEE 86*, 4 (April), 615–639.

HAUCK, S., LI, Z., AND SCHWABE, E. 1998. Configuration compression for the Xilinx xc6200 FPGA. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 138–146.

HE, C., LU, M., AND SUN, C. 2004. Accelerating seismic migration using FPGA-based coprocessor platform. In Proc. *FCCM*. 207–216.

HUDSON, R., LEHN, D., AND ATHANAS, P. 1998. A runtime reconfigurable engine for image interpolation. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 88–95.

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

KANEKO, M., YOKOYAMA, J., AND TAYU, S. 2002. 3d scheduling based on code space exploration for dynamically reconfigurable systems. In Proc. *of ISCAS*. Vol. 5, 465–468.

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

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

LIN, J.-M. AND CHANG, Y.-W. 2001. Tcg: A transitive closure graph-based representation for non-slicing floorplans. In *Proceedings of Design Automation Conference*. 764–769.

LUK, W., SHIRAZI, N., AND CHEUNG, P. Y. K. 1997. Compilation tools for runtime reconfigurable designs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*. 56–65.

MURATA, H., FUJIYOSHI, K., NAKATAKE, S., AND KAJITANI, Y. 1995. Rectangle-packing based module placement. In *Proceedings of International Conference on Computer-Aided Design*. 472–479.

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

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

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

SHOA, A. AND SHIRANI, S. 2005. Run-time reconfigurable systems for digital signal processing applications: A survey. *J. VLSI Signal Proc. 39*, 3 (March), 213–235.

SWAMINATHAN, S., TESSIER, R., GOECKEL, D., AND BURLESON, W. 2002. A dynamically reconfigurable adaptive viterbi decoder. In *Proceedings of the 10th International Symposium on Field Programmable Gate Arrays*. 227–236.

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

TESSIER, R. AND BURLESON, W. 2001. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Siganl Proc. 28*, 1 (May/June), 7–27.

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

WU, G.-M., LIN, J.-M., AND CHANG, Y.-W. 2001. An algorithm for dynamically reconfigurable FPGA placement. In *Proceedings of International Conference on Computer Design*. 501–504.

XILINX. http://www.xilinx.com/prs_rls/prs_rls2001.htm.

XILINX. 1996. XC6200 Field Programmable Gate Arrays Data Sheet. Xilinx, Inc.

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

YAMAZAKI, H., SAKANUSHI, K., NAKATAKE, S., AND KAJITANI, Y. 2000. 3d-packing by metadata structure and packing heuristics. *E83-A*, 4 (April), 639–645.

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

Received August 2005; revised March 2006, July 2006, March 2007; accepted March 2007