Floorplan Design for Multi-Million Gate FPGAs*

Lei Cheng
CS Dept & Coordinated Science Lab
University of Illinois at Urbana-Champaign
lcheng1@uiuc.edu

Martin D.F. Wong
ECE Dept & Coordinated Science Lab
University of Illinois at Urbana-Champaign
mdfwong@uiuc.edu

Abstract

Modern FPGAs have multi-millions of gates and future generations of FPGAs will be even more complex. This means floorplanning tools will soon be extremely important for the physical design of FPGAs. Due to the heterogeneous logic and routing resources on an FPGA, FPGA floorplanning is very different from the traditional floorplanning for ASICs. This paper presents the first FPGA floorplanning algorithm targeted for FPGAs with heterogeneous resources (e.g., Xilinx's Spartan3 chips consisting of columns of CLBs, RAM blocks, and multiplier blocks). Our algorithm can generate floorplans for Xilinx's XC3S5000 architecture (largest of the Spartan3 family) in a few minutes.

INTRODUCTION

Modern FPGAs have multi-millions of gates and future generations of FPGAs will be even more complex. A hierarchical approach based upon partitioning and floorplanning is necessary to successfully map a design onto an FPGA. This means FPGA floorplanning tools will soon be extremely important. (In a recent invited talk at ICCAD-2003, Dr. Salil Rajie clearly articulated the importance of the FPGA floorplanning problem as one of the multi-million gate FPGA physical design challenges [1].) A traditional FPGA design cycle consists of logical optimization [2], technology mapping [3], placement [4] and routing [5]; Fig. 1 shows the new FPGA design cycle with partitioning and floorplanning. ASIC partitioning algorithms [6] can be applied in the new FPGA design cycle. Due to the heterogeneous logic and routing resources on an FPGA, FPGA floorplanning is very different from the traditional floorplanning for ASICs. As a result, although there are many algorithms in the literature for the ASIC floorplanning [7, 8, 9, 10], these algorithms can not be used for FPGA designs. There are also some previous papers [11] on FPGA floorplanning, but all of them are targeted for older generations of FPGAs consisting only of CLBs (without RAM and multiplier blocks) where traditional ASIC floorplanning algorithms can be easily applied to solve the problem.

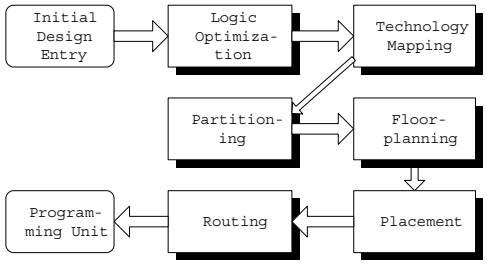

Figure 1. New Cycle for the FPGA Design

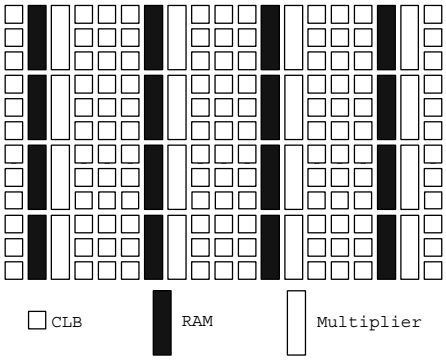

Figure 2. A simple FPGA chip

This paper is based on the following general FPGA architecture: An FPGA chip consists of columns of Configurable Logic Blocks (CLB), with column pairs of RAMs and multipliers interleaved between them. Xilinx [12] Spartan3 family and Vertex-II family conform to this architecture. For example, XC3S5000(largest chip of the Spartan3 family) consists of 80 columns of CLBs, with four column pairs of RAMs and multipliers interleaved between CLBs. In the rest of the paper, we will use a small FPGA chip as shown in Fig. 2 to illustrate the main ideas of our algorithm.

This paper presents the first FPGA floorplanning algorithm targeted for FPGAs with heterogeneous resources (e.g., Xilinx's Spartan3 chips consisting of columns of CLBs, RAM blocks, and multiplier blocks). Our algorithm can generate floorplans for Xilinx's XC3S5000 architecture in a few minutes. The algorithm is based on a non-trivial extension of the Stockmeyer floorplan optimization algorithm [13]. We use slicing structure [7] to represent floorplans, and develop an efficient algorithm that can find the optimal realization for a particular slicing structure. In this paper, we also discuss how to reduce the space complexity and how to assign irregular shapes to modules.

This paper is organized as follows. In section 2, we define the FPGA floorplanning problem. In section 3, we propose our algorithm. In section 4, we present how to compact and postprocess realizations. Section 5 presents our experimental results. We conclude our paper in section 6.

*This work was partially supported by the National Science Foundation under grants CCR-0244236 and CCR-0306244.

0-7803-8702-3/04/$20.00 ©2004 IEEE.

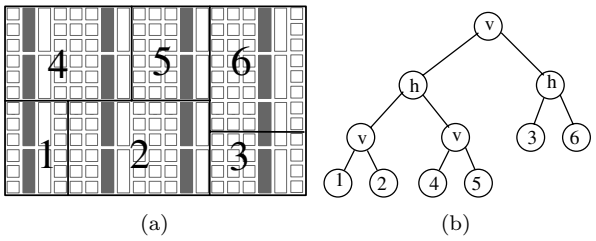

Figure 3. (a) Example floorplan and its (b) slicing tree

PROBLEM DESCRIPTION

Assume we are given a set of modules, each of them having an associated resource requirement vector $\phi = (n_1, n_2, n_3)$, which means this module requires n_1 CLBs, n_2 RAMs, and n_3 multipliers. The FPGA floorplanning problem is to place modules on the chip so that each region assigned to a module satisfies its resource requirements, regions for different modules do not overlap with each other, and a given cost function is optimized.

For example, we have 6 modules, and their resource requirement vectors are $\phi_1 = (12, 2, 1)$, $\phi_2 = (30, 4, 4)$, $\phi_3 = (15, 1, 1)$, $\phi_4 = (24, 4, 4)$, $\phi_5 = (18, 2, 2)$, $\phi_6 = (30, 2, 2)$. Fig. 3(a) is a feasible floorplan for these modules.

Actually, the floorplan in Fig. 3(a) shows a slicing structure [7]. Our algorithm uses slicing trees as representation method. A slicing floorplan is a rectangular floorplan with n basic rectangles that can be obtained by recursively cutting a rectangle into smaller rectangles. A slicing floorplan can be represented by an oriented rooted binary tree, called a slicing tree (see Fig. 3(b)). Each internal node of the tree is labelled either v or h, corresponding to either a vertical or a horizontal cut respectively. Each leaf corresponds to a basic rectangle and is labelled by the name of the module. We can use a Polish expression [14] to encode this tree, and the evaluation of this expression can be done in linear time for a traditional floorplanning problem.

In order to make our illustration easier, we would like to employ a coordinate system on the chip. In Fig. 2, the vertical unit of the coordinate system is the height of a CLB, while the horizontal unit is the width of a CLB. The lower left CLB has coordinates $(0, 0)$, the lower left RAM spans coordinates $(1, 0)$ through $(1, 2)$, and the lower left multiplier spans coordinates $(2, 0)$ through $(2, 2)$. Let H and W denote the height and the width of the chip respectively. In the rest of this paper, when we say that (x, y) is a coordinate on the chip, we always mean that x, y are integers and $0 \leq x < W, 0 \leq y < H$.

A rectangle $r = (x, y, w, h)$ on the chip is characterized by its lower left coordinate (x, y), its width w and its height h. $x(r)$, $y(r)$, $w(r)$, $h(r)$ denote the corresponding fields of r. In Fig. 3(a), the rectangle labelled 1 is $r_1 = (0, 0, 4, 6)$. Given a rectangle r, we use ϕ_r to denote the resource vector associated with r, for example $\phi_{r_1} = (12, 2, 2)$. Let $\mathcal{R}$ denote the set of all possible rectangles on the chip. Given a rectangle $r \in \mathcal{R}$, $x(r) + w(r) \leq W$ and $y(r) + h(r) \leq H$. We need several definitions first before we can continue.

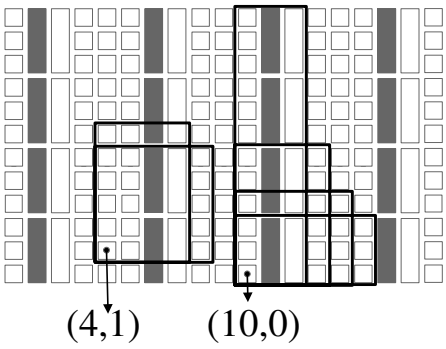

Figure 4. Irreducible realization lists

Definition 1 Resource vectors comparison: Given two resource vectors $\phi = (n_1, n_2, n_3)$ and $\phi' = (n'_1, n'_2, n'_3)$, the inequality $\phi \leq \phi'$ holds iff $n_1 \leq n'_1 \wedge n_2 \leq n'_2 \wedge n_3 \leq n'_3$.

Definition 2 Given a module θ with a resource requirement vector ϕ, the realization set of θ is $\mathcal{R}_\theta = \{r \mid r \in \mathcal{R}, \phi \leq \phi_r\}$.

Definition 3 Given two rectangles $r_1 = (x, y, w_1, h_1)$ and $r_2 = (x, y, w_2, h_2)$, r_1 dominates $r_2 (r1 \prec r2)$ iff $w_1 \leq w_2 \wedge h_1 \leq h_2$.

Intuitively, realization set $\mathcal{R}_\theta$ is the set of rectangular regions that satisfy the resource requirement of module θ. r_1 dominating r_2 means r_2 is redundant. Obviously, there are many redundant realizations in $\mathcal{R}_\theta$, so we introduce the following notation.

Definition 4 Irreducible Realization List(IRL) for module θ starting from coordinate (x, y) is defined as $\mathcal{L}(\theta, x, y) = \{r \mid r \in \mathcal{R}_\theta, x(r) = x \wedge y(r) = y$, and no other $r' \in \mathcal{R}_\theta$ dominates $r\}$.

IRLs of a module are different for different starting points. For example, assume that $\phi = (12, 1, 1)$ is the resource requirement vector of module θ. Its IRLs starting from $(4, 1)$ and $(10, 0)$ are $\mathcal{L}(\theta, 4, 1) = \{(4, 1, 4, 6), (4, 1, 5, 5)\}$ and $\mathcal{L}(\theta, 10, 0) = \{(10, 0, 3, 12), (10, 0, 4, 6), (10, 0, 5, 4), (10, 0, 6, 3)\}$ (see Fig. 4). We can see from this example that there is a fundamental difference between the FPGA floorplanning problem and the traditional floorplanning problem.

By definition, any two realizations in an IRL can't dominate each other. If we sort the list according to $h(r)$ from high to low, then $w(r)$ must be sorted from low to high. An IRL is always sorted this way in the rest of this paper. We have the following lemma regarding IRLs:

Lemma 1 Assume we are given a module θ and a rectangle $r = (x, y, w, h) \in \mathcal{R}_\theta$. If $x' \leq x$ and $y' \leq y$, then there is a rectangle $r' \in \mathcal{L}(\theta, x', y')$, such that $x(r')+w(r') \leq x(r)+w(r)$ and $y(r')+h(r') \leq y(r)+h(r)$ (see Fig. 5).

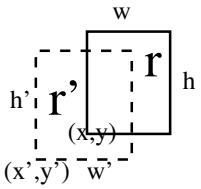

Figure 5. Figure for Lemma 1

FPGA FLOORPLAN ALGORITHM

Roughly speaking, we use slicing trees to represent an FPGA floorplan and perturb slicing trees under the control of simulated annealing in a way similar to [14]. Given a slicing tree, we calculate IRLs of each node from leaves to the root. Then we check IRLs of the root. If there is a realization $r \in \mathcal{L}(root, 0, 0)$ satisfying $w(r) \leq W$ and $h(r) \leq H$, we find a feasible solution for the FPGA floorplanning problem.

Now, we are extending the definition of IRL to include tree nodes. Given two rectangles $r_1 = (x_1, y_1, w_1, h_1)$ and $r_2 = (x_2, y_2, w_2, h_2)$, the bounding rectangle of r_1 and r_2 is a rectangle $r = (x, y, w, h)$, where $x = min\{x_1, x_2\}$, $y = min\{y_1, y_2\}$, $w = max\{w_1 + x_1, w_2 + x_2\} - x$ and $h = max\{h_1 + y_1, h_2 + y_2\} - y$. Given a tree node u, if u represents a module θ, $\mathcal{R}_u = \mathcal{R}_\theta$. If u is an internal node, let v, q be the left and the right child of u. If u is vertical, $\mathcal{R}_u$ consists of all bounding rectangles of $r_1 \in \mathcal{R}_v$ and $r_2 \in \mathcal{R}_q$, where r_1 is to the left side of r_2; if u is horizontal, $\mathcal{R}_u$ consists of all bounding rectangles of $r_1 \in \mathcal{R}_v$ and $r_2 \in \mathcal{R}_q$, where r_1 is below r_2. The irreducible realization list for a tree node u is defined as $\mathcal{L}(u, x, y) = \{r \mid r \in \mathcal{R}_u, x(r) = x \wedge y(r) = y$, and no other $r' \in \mathcal{R}_u$ dominates $r\}$.

In the next subsection, we will talk about how to compute IRLs of a node efficiently if we know IRLs of its children. Then we will present how to reduce the space complexity. In the third subsection, we present formal algorithms. Finally, we introduce our cost function and some implementation details that can improve the performance.

Computing Irreducible Realization Lists

It is obvious that we only need to calculate IRLs once for basic modules. We precompute them at the beginning of simulated annealing. Experiments show that this part only takes a very small amount of runtime.

As we have said, we calculate IRLs for each node from leaves to the root. Let u denote an internal node which has a vertical cut, v and q denote the left and the right child of node u. We have the following lemma:

Lemma 2 If $r \in \mathcal{L}(u, x, y)$, then there exist r_1, r_2 such that $r_1 \in \mathcal{L}(v, x, y), r_2 \in \mathcal{L}(q, x + w(r_1), y)$, and r is an bounding rectangle of r_1 and r_2.

Proof:

Assume $r = (x, y, w, h)$ is an bounding rectangle of realizations $r_1 = (x_1, y_1, w_1, h_1) \in \mathcal{L}(v, x_1, y_1)$ and $r_2 = (x_2, y_2, w_2, h_2) \in \mathcal{L}(q, x_2, y_2)$ that do not satisfy the lemma. We prove that there exist r_1' and r_2' that satisfy the lemma and have r as their bounding rectangle. There are six combinations of r_1, r_2 to generate r as shown in Fig. 6.

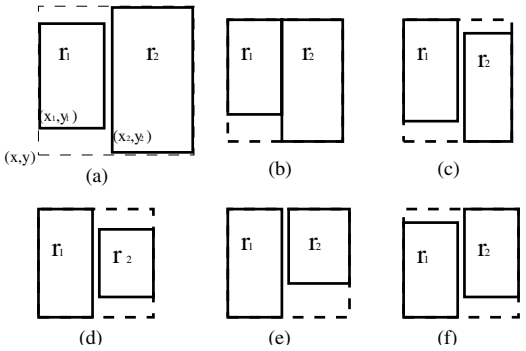

Figure 6. Six combinations

First we consider case (a) where $x = x_1, y = y_2$. Because r_1 is a realization for v, we know by Lemma 1 that there must exist a realization $r_1' = (x, y, w', h') \in \mathcal{L}(v, x, y)$, $w' \leq w_1$, and $y + h' \leq y_1 + h_1$. Now it is true that $r_2' = (x + w', y, w - w', h) \in \mathcal{L}(q, x + w', y)$, or $r \in \mathcal{L}(u, x, y)$ will be a contradiction.

The correctness of the other 5 cases can be proved in a similar way. □

This lemma tells us that a realization $r \in \mathcal{L}(u, x, y)$ of an internal node u can be generated by two horizontally aligned realizations of its children and there is no horizontal gap between them. With this lemma, we know that it is enough to combine every realization $r \in \mathcal{L}(v, x, y)$ with realizations of $\mathcal{L}(q, x + w(r), y)$ to generate $\mathcal{L}(u, x, y)$. The heights of realizations in an IRL are sorted from high to low; their widths are sorted from low to high. Assume $\mathcal{L}(v, x, y) = \{r_1, r_2, \ldots, r_s\}$ is sorted as expected. When we combine a realization r_i with realizations of q, we do not need to consider all combinations. For those realizations of q with heights not larger than $h(r_i)$, we only need to consider the highest one to get a minimum width (Fig. 7(a)). We also do not need to combine r_i with a realization r' of q if $h(r') \geq h(r_{i-1})$[1]. Let's refer to Fig. 7(b). We can see from the figure that if $h(r') \geq h(r_{i-1})$, there must exist a realization $r_1' \in \mathcal{L}(q, x + w(r_{i-1}), y)$ by Lemma 1, such that $h(r_1') \leq h(r')$ and $w(r_{i-1}) + w(r_1') \leq w(r_i) + w(r')$, which means the bounding rectangle of r_{i-1} and r_1' dominates that of r_i and r'. So we do not need to consider those realizations of the right child with heights no less than $h(r_{i-1})$. Let l denote $max\{H, W\}$. With the above analysis, we can prove the following theorem.

Theorem 1 For an internal node u, and a coordinate (x, y), $\mathcal{L}(u, x, y)$ can be constructed in $O(l \log l)$ runtime.

Proof:

We follow the notations we use in the above paragraph in

[1] Assume $h(r_0) = H + 1$.

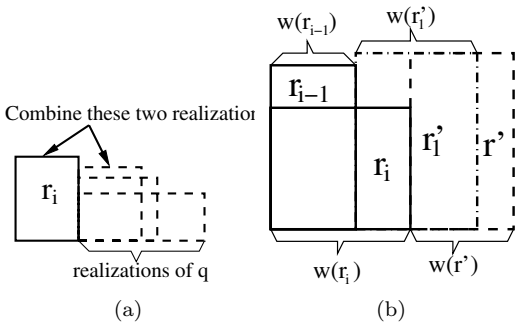

Figure 7. Redundant combinations

this proof, such as v, q are the left and the right child of u.

First we consider the case in which u is a vertically sliced node. When we combine r_i with realizations of q, we get a set of realizations with heights between $h(r_i)$(may be inclusive) and $h(r_{i-1})$(exclusive). For each i, we first find the highest realization $r' \in \mathcal{L}(q, x + w(r_i), y)$ with height not larger than $h(r_i)$, then we scan $\mathcal{L}(q, x + w(r_i), y)$ in reverse order until we reach a realization with height not less than $h(r_{i-1})$. During this time, we scan at most $h(r_{i-1}) - h(r_i)$ realizations. We can design a data structure so that find operation costs $O(\log H)$(the length of the list is at most H), and single scan operation costs $O(1)$, so the runtime for generating $\mathcal{L}(u, x, y)$ is

$$\begin{aligned} & \sum_{i=s}^{1} O(h(r_{i-1}) - h(r_i) + \log H) \\ = \; & O(h(r_0) - h(r_s) + s \log H) \\ = \; & O(l \log l) \end{aligned}$$

For horizontally sliced node, we have a similar proof. □

Figure 8. Pattern of Fig. 2

Taking Advantage of Repetition

If we implement the previous algorithm directly on the chip, finding IRLs for every coordinate will make the space complexity formidable. Fortunately, real FPGA chips are very regular. Each chip is a repetition of a basic pattern. Consider the example chip in Fig. 2, of which basic pattern is shown in Fig. 8. It turns out that slicing can gracefully utilize this repetition property, and we only need to perform computation on the pattern instead of the whole chip. This is the most important reason that we use slicing in our algorithm.

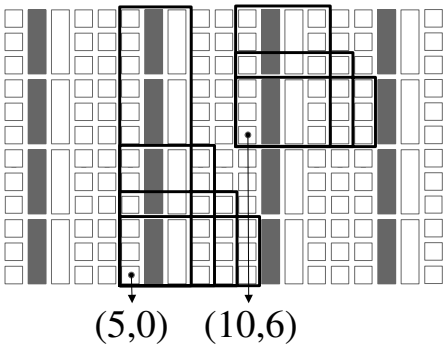

Figure 9. Example for Observation 2

Let h_p and w_p denote the height and the width of the pattern. We have the following observations.

Observation 1 Given a module θ and its two irreducible realization lists $\mathcal{L}_1 = \mathcal{L}(\theta, x_1, y_1), \mathcal{L}_2 = \mathcal{L}(\theta, x_2, y_2)$ where $x_2 = x_1 + n_1 * w_p, y_2 = y_1 + n_2 * h_p$, n_1 and n_2 are two integers[2]. $r_1 \in \mathcal{L}_1, r_2 \in \mathcal{L}_2$, we have $w(r_1) = w(r_2) \iff h(r_1) = h(r_2)$.

Assume (x_1, y_1) and (x_2, y_2) map to the same coordinate on the pattern, and $r_1 \in \mathcal{L}(\theta, x_1, y_1), r_2 \in \mathcal{L}(\theta, x_2, y_2)$, then r_1 and r_2 have the same shape if $w(r_1) = w(r_2)$ or $h(r_1) = h(r_2)$. Given a realization r, $h(r) \leq H - y(r)$ and $w(r) \leq W - x(r)$. Let $\mathcal{S}(\mathcal{L})$ denote the shape set of $\mathcal{L}$, $\mathcal{S}(\mathcal{L}) = \{(w(r), h(r)) \mid r \in \mathcal{L}\}$. Together with the above observation, we have:

Observation 2 Given a module θ, two points (x_1, y_1) and (x_2, y_2) where $x_2 = x_1 + n_1 * w_p, y_2 = y_1 + n_2 * h_p, n_1 \geq 0$ and $n_2 \geq 0$ are two integers, then it must be the case that $\mathcal{S}(\mathcal{L}(\theta, x_2, y_2)) \subseteq \mathcal{S}(\mathcal{L}(\theta, x_1, y_1))$

Fig. 9 shows an example. In this chip layout, $w_p = 5$ and $h_p = 3$. Given two points $(x_1, y_1) = (5, 0)$, $(x_2, y_2) = (10, 6)$, and a module with resource requirement vector $\phi = (12, 1, 1)$. We know from the figure that $\mathcal{S}(\mathcal{L}(\theta, x_1, y_1)) = \{(6, 3), (5, 4), (4, 6), (3, 12)\}$, and $\mathcal{S}(\mathcal{L}(\theta, x_2, y_2)) = \{(6, 3), (5, 4), (4, 6)\}$. So $\mathcal{S}(\mathcal{L}(\theta, x_2, y_2)) \subseteq \mathcal{S}(\mathcal{L}(\theta, x_1, y_1))$.

In our algorithm, we calculate an IRL on every coordinate of the pattern for every module. The widths and heights of these realizations range from 1 to W and 1 to H respectively. We use IRLs on the pattern to represent those on the chip. Even though this may introduce some illegal realizations (say $x + w > W$), we allow these illegal realizations and pay a penalty in the objective function.

Algorithm

Algorithm Get_Realization_list_V computes a realization list for an internal vertically sliced node u. We calculate IRLs for every point on the pattern separately. This algorithm takes a node u and a point (x, y) as its input parameters, and calculates the IRL of u starting from (x, y).

[2] $(x_1, y_1), (x_2, y_2)$ map to the same coordinates on the pattern.

Get_Realization_list_V(u, x, y)
Begin:
    $\mathcal{L}(u, x, y) \leftarrow \emptyset$ /*initially empty*/
    $l_v \leftarrow \mathcal{L}(v, x, y)$
    len $\leftarrow$| l_v | /*length of l_v*/
    for i:=len to 1
        $x_q = (x + w(l_v[i])) \bmod w_p$
        Let l_q be $\mathcal{L}(q, x_q, y)$
        if i= 1
            upperheight $\leftarrow \alpha * H + 1$
        else
            upperheight $\leftarrow h(l_v[i-1])$
        find j, satisfying $h(l_q[j]) \leq h(l_v[i])$
                and $h(l_q[j-1] > h(l_v[i])$ [3]
        while (j ≥ 1 and $h(l_q[j]) < upperheight$) do
            $h_{new} \leftarrow max(h(l_q[j]), h(l_v[i]))$
            $w_{new} \leftarrow w(l_q[j]) + w(l_v[i])$
            if ($\mathcal{L}(u, x, y)$ is empty and $w_{new} < \alpha * W$)
                    or $w_{new} <$ width of the first element in
                        $\mathcal{L}(u, x, y)$
                $r_{new} = (x, y, w_{new}, h_{new})$
                insert r_{new} as the first element to $\mathcal{L}(u, x, y)$
            j $\leftarrow$ j-1
End

In this algorithm, $\alpha \geq 1$ is a constant, and we allow solutions with areas as large as $\alpha * W * \alpha * H$ as our intermediate solutions. Algorithm Get_Realization_list_H for horizontally sliced nodes is very similar to Get_Realization_list_V. We can prove the following theorem, which tells us that the algorithm can always find the optimal solution for a slicing structure.

Theorem 2 Given irreducible realization lists of basic modules, algorithm Get_Realization_list_V always computes the complete irreducible realization list with respect to α, no more no less.

The algorithm for evaluating a slicing tree is presented below.

Evaluate_Tree(T)
    Evaluate_Node(T.root)

Evaluate_Node(u)
    if u is leaf return
    Evaluate_Node($u.left$)
    Evaluate_Node($u.right$)
    for every point (x, y) on the pattern do
        if u is vertically sliced
            Get_Realization_list_V(u,x,y)
        else
            Get_Realization_list_H(u,x,y)

Let l denote $max(H, W)$, m denote the number of modules, p denote the number of points on the pattern, that is $p = h_p w_p$. The length of an IRL can not be longer than αl.

[3]We can insert pivot values to both ends of l_r to guarantee the existence of j.

From Theorem 1, the runtime of Get_Realization_list_V is $O(\alpha l \log(\alpha l)) = O(l \log(l))$. Since a slicing tree is a full binary tree, the number of internal nodes is $m - 1$. For each internal node, we compute its p IRLs, so we have the following theorem.

Theorem 3 The complexity of evaluating a slicing tree is $O(mlp \log l)$. And the algorithm needs $O(mlp)$ memory space.

Several Implementation Details

We employ following ideas to make our implementation much more efficient.

1. Cost Function
   $C = \alpha Area + \beta Ratio\_Sum + \gamma Wire\_Length$
   We consider area as well as wire length in our cost function. In this cost function, the area is the addition of two parts: the area of the enclosed rectangle of the floorplan, and the area that are not inside the chip for an illegal floorplan. Wire length also consists of two parts, internal wire length inside each module and external wire length between different modules. For external wire length, we use traditional half perimeter of bounding box method. The internal wire length can be estimated by aspect ratio of the corresponding module. A smaller aspect ratio tends to decrease the longest distance between any two devices inside the module. The smaller the aspect ratio, the smaller is the internal wire length. For each realization r_θ of module θ, it contributes $\frac{w(r_\theta)}{h(r_\theta)} + \frac{h(r_\theta)}{w(r_\theta)}$ to $Ratio\_Sum$. So far, we have not considered IOBs in this paper. We believe we can add another term χ to the cost function so that the connections between modules and IOBs are also optimized. $\chi = \sum d_i IOB_i$, where d_i is the distance between module θ_i and its nearest boundary of the chip, and IOB_i is the number of IOBs module θ_i needs. In the cost function, α, β and γ are used to control the relative importance of $Area$, $Ratio\_Sum$ and $Wire\_Length$.
2. When we calculate the IRLs for basic modules, we only consider those realizations with relatively small aspect ratios. This technique can greatly reduce the length of IRLs, thus improving the runtime.
3. We use a traditional floorplanner to generate a better initial configuration for simulated annealing. To use a traditional floorplanner, every module should have dimensional information. We generate a rectangular shape for each module according to resources it needs. By doing this, we can start from a relatively lower temperature, so that we can save runtime.

COMPACTION AND POSTPROCESSING

Rectangular realizations may waste resources. A rectangular realization may contain more resources than needed, so a solvable floorplanning problem may not be solved if we only allow rectangular realizations. Assume we want to place

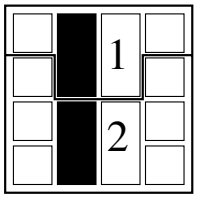

Figure 10. An example for irregular realizations

two modules on the small chip in Fig. 10. The resource requirement vectors of these modules are $\theta_1 = (2, 1, 1)$ and $\theta_2 = (6, 1, 1)$. These two modules can only be placed on the chip if we allow irregular realizations. In this section, we will describe how to assign a rectilinear realization to a module. The way we deal with it is compaction and postprocessing.

Compaction

After evaluating a slicing tree, we check every realization $r \in \mathcal{L}(root, 0, 0)$. If $w(r) \leq W \wedge h(r) \leq H$, we find a feasible solution. If only $w(r) \leq W$, we compact r vertically using the following algorithm.

1. Compute the lower left coordinate of every module on the chip according to r.
2. Sort modules according to y value of the lower left coordinate from low to high. Let module list $(\theta_1, \theta_2, \ldots, \theta_m)$ be the sorted result.
3. Set contour line C_1 to be the lower boundary of the chip.
4. `for` $i = 1$ `to` m `do`
    1. Place module θ_i on the contour line C_i without changing the x-coordinate of θ_i.
    2. Let C_{i+1} be the upper contour line after we place θ_i on C_i (see Fig. 11).

Fig. 11 shows that θ_i is pushed into a concave part of C_i. In our implementation, we can control how deep a module can be pushed into such a concave.

Postprocessing

There are several disadvantages of compaction. After compaction, some modules are placed in bad shapes, and there exists large amount of white space on the top of the chip.

We use a postprocessing technique to fix these problems. Let's revisit our compaction algorithm. Just before we compact module θ_m, all other modules are placed between the contour line C_m and the lower boundary of the chip, so we can place the last module freely between C_m and the upper boundary of the chip. We call the upper boundary of the chip the upper contour line C'. We use some heuristic method to find a good place between C_m and C' for module θ_m near the upper contour line, and update the upper contour line C' according to the placement of θ_m. Then we place module θ_{m-1} between C_{m-1} and the new upper contour line C'. Generalizing this process, we have the following algorithm:

1. Set the upper contour line C' to be the upper boundary of the chip.

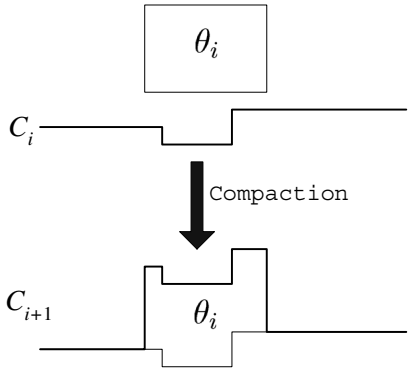

Figure 11. Update of a contour line

2. `for` $i = m$ `downto` 1 `do`
    a. Use a heuristic method to find a good place for module θ_i near C' between C' and C_i.
    b. Adjust the upper contour line C' according to the placement of the module.

We only invoke this postprocessing when the placement after compaction can fit into the chip. It is easy to see that if the placement after compaction can fit into the chip, then the placement after postprocessing can fit into the chip. Experiments show that this postprocessing really improves module shapes. Of course, this postprocessing does not eliminate unoccupied space, but it helps to distribute the unoccupied space evenly on the chip, which is helpful for routing and heat dissipation. We use the following heuristic to find a good place for module θ_i between two contour lines C' and C_i. Let S_i be the area of θ_i, $S_i = n_1 s_1 + n_2 s_2 + n_3 s_3$, where (n_1, n_2, n_3) is the resource requirement vector of θ_i, and s_1, s_2, s_3 are the areas of one CLB, RAM, multiplier respectively. Let $d_i = \sqrt{S_i}$. Let $C[i]$ denote y value of the coordinate on C with x value equaling to i. Given $1 \leq j \leq k \leq W$, let $\rho(C, j, k) = min\{C[t] \mid j \leq t \leq k\}$. Our heuristic will try to find $1 \leq j \leq k \leq W$ such that $|k - j - d_i|$ is minimized and the resources in region $R(j, k)$ satisfying the resource requirement of θ_i, $R(j, k) = \{(\hat{x}, \hat{y}) \mid j \leq \hat{x} \leq k, \hat{y} \geq C_i[\hat{x}], \hat{y} \leq C'[\hat{x}], \hat{y} \leq \rho(C', j, k) + \lambda d_i\}$. λ is used to control the irregularity of the placement. The larger the λ, the more the irregularity. Then we place module θ_i on the top of region $R(j, k)$.

EXPERIMENTAL RESULTS

Our experiments are carried out on a desktop with a 2.4GHz Intel(R) Xeon(TM) CPU. The OS is Red Hat Linux 8.0, and we use g++ to compile our programs. We test our program on Xilinx XC3S5000 FPGA, which is the largest FPGA in the Spartan-3 family. XC3S5000 has 8320 CLBs, 104 RAMs, and 104 multipliers. Parameter values of this FPGA are 104 and 352 for l, p respectively.

The first experiment we do is to divide the XC3S5000 almost evenly into 20 blocks. So we have 20 modules. 16 modules need 400 CBLs, 5 RAMs and 5 multipliers each, and the

Table 1. Testing data

Name	#modules	CLB percentage(%)	RAM percentage(%)	Multiplier percentage(%)	Runtime(s)	
					Slicing	With compaction
FPGA1	21	72%	88%	86%	59	1
FPGA2	23	72%	84%	84%	27	2
FPGA3	21	80%	75%	75%	30	1
FPGA4	23	81%	73%	73%	16	1
FPGA5	23	83%	81%	81%	63	4
FPGA6	37	94%	75%	75%	Fail	173
FPGA7	50	78%	78%	76%	88	28
FPGA8	100	78%	79%	77%	242	40

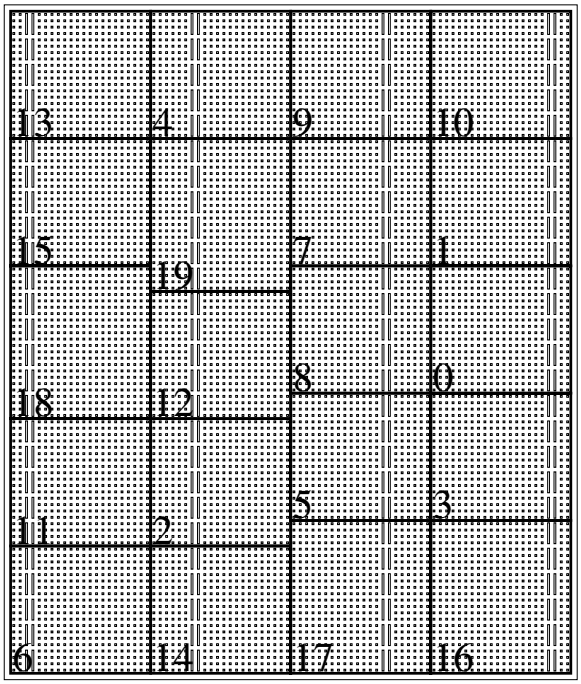

Figure 12. Result of a 20-module problem obtained in 88 s

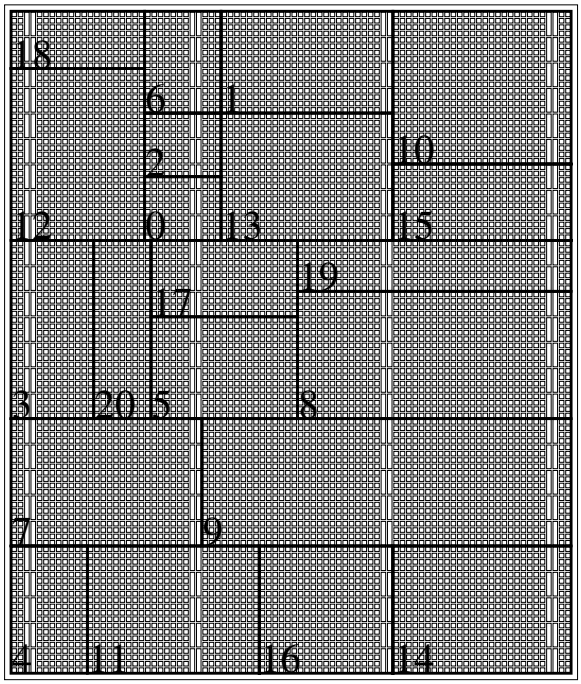

Figure 13. Floorplan result of FPGA1 obtained in 59 s

other four modules need 480 CBLs, 6 RAMs, and 6 multipliers each. We run our program on these 20 modules, and get the result shown by Fig. 12 in 88 seconds. This is a very tight problem and our floorplanner is able to reconstruct the floorplan.

We also run our algorithm on 32 testing data. Resource utilizations of these data range from 70% to 95%. With slicing alone, we can solve 24 of them. With compaction and postprocessing, we can solve all of them. Some of our test results are shown in Table 1. The failure of FPGA6 by slicing alone is due to the large requirement of CLBs. Fig. 13 is the floorplan result for 21 modules using slicing alone. The total resources needed by these 21 modules are 6000 CLBs (72.12% of total CBLs), 91 RAMs (87.5% of total RAMs), and 89 multipliers (85.58% of total multipliers). It takes 59 seconds to get the result. Fig. 14 shows a floorplan with compaction and postprocessing, the unnumbered areas in this figure are empty spaces. There are no empty spaces shown in Fig. 13 because we have distributed them into realizations.

Even though compaction and postprocessing need additional time to evaluate a particular slicing tree, the runtime of our algorithm with compaction and postprocessing is much less than that without compaction. The reason is that simulated annealing experiences a lot of neighborhood permutations (i.e., slicing tree changes) before the algorithm finds a feasible solution, and the compaction can greatly decrease the number of such neighborhood movements because many unfeasible slicing trees (a slicing tree does not correspond to any feasible solution) may produce a feasible solution after compaction.

To the best of our knowledge, there are no FPGA floorplanning tools available for the heterogeneous resources, even

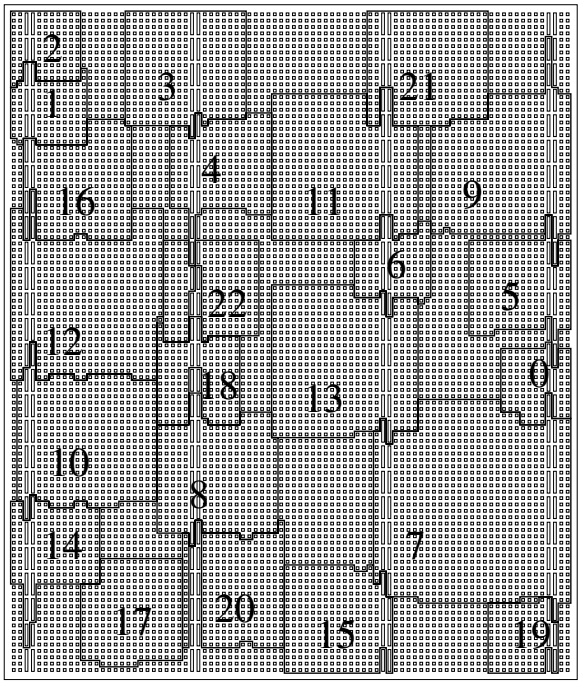

Figure 14. Floorplan result for FPGA2 with compaction

though there are many FPGA placement tools. So we can't compare our method with others. The experimental results show that our algorithm is both efficient and effective, and it can produce a feasible solution in a few minutes for an input data with reasonable resource requirements. The experimental results also indicate that our algorithm works very well on large data. (In a floorplanning problem, the number of modules is far less than 100 in most cases.) Actually, the scalability of our algorithm is guaranteed by Theorem 3, since the runtime is linear to the number of modules.

CONCLUDING REMARKS

In this paper, we present the first FPGA floorplanning algorithm targeted for FPGAs with heterogeneous resources. We use slicing representation, and compute irreducible realization lists for every node of the slicing tree on a pattern of the chip. There is no redundance inside each IRL, but there may exist redundances between different IRLs of a module. These redundances are important for our fast algorithm. Without these redundances, lemma 2 (which is the basis of our algorithm) will not be true. If lemma 2 is not true, we will have to combine every realization of the left child with every realization of the right child to get the realization set of a node, and it is really time consuming to do so. In this paper, slicing is used because it works very well with the pattern property of an FPGA chip, and we have no idea whether other floorplan representations can make such good use of this property.

Slicing alone does not always guarantee a feasible solution, so we introduce compaction and postprocessing. Intuitively, compaction pushes every realization down on the chip, and a feasible solution may be found after this process. With compaction, the runtime of our algorithm is greatly shortened. Postprocessing technique is used to improve the placement shapes of modules (i.e., make them more squarish), and it's also helpful to distribute white space more evenly onto the chip. Experimental results show that our algorithm is efficient and effective.

REFERENCES

[1] Maogang Wang, Abhishek Ranjan, and Salil Raje. Multi-million gate FPGA physical design challenges. ICCAD, pages 891–898, 2003.

[2] D. Gregory, K. Bartlet, A. De Geus, and G. Hachtel. Socrates: a system for automatically synthesizing and optimizing combinational logic. DAC, pages 79–85, 1996.

[3] Robert Francis, Jonathan Rose, and Zvonko Vranesic. Chortle-crf: fast technology mapping for lookup table-based FPGAs. DAC, pages 227 – 233, 1991.

[4] C. Sechen and K. Lee. An improved simulated annealing algorithm for row-based placement. ICCAD, pages 478–481, 1987.

[5] J. Rose and S. Brown. Flexibility of interconnection structures in field-programmable gate arrays. IEEE Journal of Solid State Circuits, 26(3):277–282, March 1991.

[6] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. DAC, pages 526 – 529, 1997.

[7] Ralph H. J. M. Otten. Automatic floorplan design. DAC, pages 261–267, 1982.

[8] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module palcement. ICCAD, pages 472–479, 1995.

[9] X. Hong et al. Corner Block List: an effective and efficient topological representation of non-slicing floorplan. ICCAD, pages 8–11, 2000.

[10] E. F. Y. Yong, C. C. N. Chu, and Z. C. Shen. Twin Binary Sequences: a nonredundant representation for general nonslicing floorplan. TCAD, 22(4):457–469, April 2003.

[11] J. M. Emmert and D. Bhatia. A methodology for fast FPGA floorplanning. International Symposium on Field Programmable Gate Arrays, 1999.

[12] http://www.xilinx.com.

[13] Larry J. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. Information and Control, 57(2/3):91–101, 1983.

[14] D. F. Wong and C. L. Liu. A new algorithm for floorplan design. DAC, pages 101–107, 1986.