

Physical Design for Reconfigurable Computing Systems using Firm Templates *

R. Kastner K. Bazargan M. Sarrafzadeh
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118
{kastner,kiarash,majid}@ece.nwu.edu

Abstract

The advances in the programmable hardware has lead to new architectures where the hardware can be dynamically adapted to the application to gain better performance. There are still many challenging problems to solve before any practical general-purpose reconfigurable system is built. One fundamental problem is the placement of the modules on the reconfigurable functional unit (RFU). Here, we are interested in offline placement in which the schedule is known at compile time. In this paper we present firm templates, a module with limited shape and flexibility, which leads to more compact placements. The firm template is a natural and effective compromise between hard and soft templates. A hard template has only one shape, therefore the RFUOP must be placed as that shape. On the other end of the spectrum, a soft template can assume any shape. Obviously, a soft template would yield a fully packed reconfigurable unit. Unfortunately, it takes a tremendous amount of time to reshape an RFUOP since you must physically redesign the operation to fit the given area and shape. Experimental results show that firm templates provide 43.25% better compaction than hard templates.

1 Introduction

As the FPGAs get larger and faster, both the number and complexity of the modules which can be loaded onto them increase, hence better speedups can be achieved by exploiting FPGAs in hardware systems. Gokhale *et. al.* report speedups of 200x in [5] for the string matching problem (i.e., the program

runs 200 times faster when run on the FPGA board than when run on a Sparc machine). Furthermore, the ability to partially reconfigure the chip as it is running, enables the implementation of dynamically reconfigurable hardware systems which adapt themselves to the application for better performance [5, 9, 14]. Hauck has reported many applications for reconfigurable systems in [7]. Such systems usually consist of a host processor and an FPGA “co-processor” called Reconfigurable Functional Unit (RFU) which can be programmed *in the course of the running time of the program* with varying configurations at different stages of the program. An example is shown in Figure 1. Figure 1-a, shows three parts of the code which are mapped to RFU operations (or RFUOPs, also called modules). When the program is running the loop containing RFUOP2 (time $t1$ in Figure 1-a) two RFUOPs are loaded on the chip. Later on, when the program is about to enter the loop at time $t2$, there is no space on the RFU to place RFUOP3; hence RFUOP2 is swapped out of the chip and RFUOP3 is loaded. RFUOP1 is still on the chip and may be invoked later in the program.

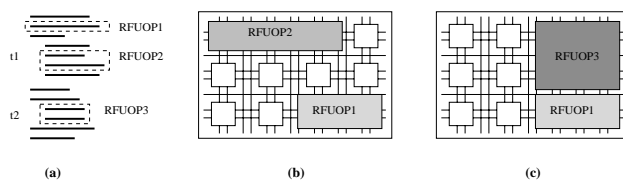


Figure 1: (a) The running code (b) RFU configuration at time $t1$ (c) RFU configuration at a later time $t2$

Unfortunately, rather long delays in reprogramming RFUs keeps us from achieving very high speedups in general purpose computing. Wirthlin and Hutchings [14] report an overall speedup of 23x, while the

*This work was supported by DARPA under contract number DABT63-97-C-0035.

speedup could be 80x if configuration time was zero (the configuration time is 16% to 71% of the total running time).

A number of methods have been proposed to overcome the delays in reconfiguring the RFUs, e.g., [6, 8]. Although these algorithms are necessary for a practical reconfigurable system, we still need fast and powerful physical design CAD tools to do RFU real estate management both offline and online. In the offline version, the flow of the program is known in advance (e.g., in DSP applications, or loops containing basic blocks) and hence the configuration management component can do various optimizations in the configuration of the RFU before the system starts running. On the contrary, in the online version the decision on what operations should be launched is not known *a priori*. The flow of the program is not known in advance and hence the RFU configuration management should be done on the fly.

Both online and offline versions of the placement algorithms are important for reconfigurable computing systems. Due to the hard nature of accurately predicting the run time behavior of a general program at compile time, one needs online placement algorithms for at least parts of the RFU manager kernel. The offline algorithm can be exploited to generate compact placements for a group of RFU operations which will execute in sequence, e.g., part of the code in a basic block (the compact placement of the group of RFU modules can be seen as one atomic module when the online placement method is running). Furthermore, placements generated by an offline method can serve as baseline solutions for the online versions, and help us devise better online algorithms. Hence, the most important feature of an offline placement algorithm is the quality of the placement it generates, even though it is a slow method.

To this date, the place and route algorithms proposed for FPGAs are generally very slow or do not generate high quality placements. Examples are [10, 11, 12]. The only fast placement algorithm reported in the literature is a work by Callahan *et. al.* [4] which is a linear time algorithm for mapping and placement of data flow graphs on FPGAs. Their algorithm utilizes the FPGA area efficiently, but it is limited to datapaths only.

Our goal is to devise efficient methods for placing RFU operations on the chip as compactly as possible so that the results can be used both by the offline algorithm and as a baseline for assessing the quality of online methods. We propose firm templates for offline placement of the modules on RFU, and show the effec-

tiveness of the proposed methods by comparing their placements with other offline algorithms (See [1, 15]).

The rest of the paper is organized as follows: In Section 2 we have described our model of the reconfigurable system. We have also defined measures to compare effectiveness of different RFUOP placement algorithms. Our methods are described in Section 3 and 4. Experimental results are shown in Section 5. Section 6 contains a conclusion and discussion on possible ways to improve our algorithms and further experiments to give us more insight on the nature of the problem.

2 Our Model of a Reconfigurable Computing System

Brebner [2] suggests an environment in which the runtime system dynamically chooses between hardware (RFU operation) and software (main host CPU instructions) implementations of the same function based on profile data or other criteria. We use the same paradigm in our model. An RFUOP r_i can be either *accepted* or *rejected* based on availability of RFU real estate. If an RFUOP is rejected, the same function should be performed by the host CPU and hence a running time penalty is incurred. We use set \mathcal{ACC} to represent RFUOPs which are accepted (See Equation 1).

Our model which deals with the placement engine of the RFU configuration management interface, assumes that RFUOPs have been scheduled during compile time. Furthermore, it does not consider any caching of the modules on the chip during the runtime.

The set

$$\mathcal{RFUOPS} = \{r_1, r_2, \dots, r_n \mid r_i = (w_i, h_i, s_i, e_i)\}$$

represents all the RFU operations defined in the system, where w_i, h_i, s_i and e_i are all positive integers with the additional constraint that $s_i < e_i$. w_i and h_i are the width and height of the implementation of the RFUOP r_i in the library respectively. s_i is the time the operation r_i is invoked and $(e_i - s_i)$ is the time-span it is resident in the system.

The placement engine can be invoked in only two ways: *insert* a module which is not currently on the chip (at time s_i) and *delete* a currently placed module from the chip (at time e_i). If there is a cache manager in the system (See Figure 2), it will issue insertion/deletion requests to the placement engine only when such operations should actually take place. For

example if an RFUOP is invoked and the cache manager detects that the module is already on the chip, it will issue no requests to the placement engine. On the other hand, if a module which was previously swapped out (placement engine had received a *delete* command on that RFUOP) and is invoked again, the cache manager will request the placement engine to insert the RFUOP as if it was the first time this RFUOP is being invoked.

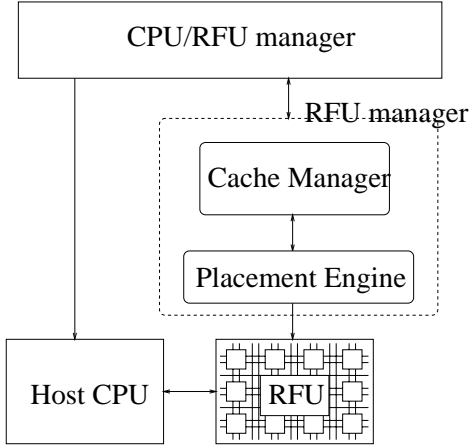


Figure 2: A sample model of a reconfigurable computing system

At any given time, there might be a number of modules on the RFU which can perform different operations concurrently. If in such a case a new RFUOP is invoked (cache manager sends an *insert* request to the placement engine) and there is no space and no idle RFUOP on the chip, then the request is *rejected*. Since the RFU cannot perform the operation, the main CPU should execute instructions to perform the same function, incurring some penalty to the running time. Otherwise (if the RFUOP is *accepted*), it is loaded onto RFU and executed. We assume that higher levels of the RFU configuration management will block insertion requests for RFUOPs which have not shown performance gains, i.e., the application profile data shows that the time to load the RFUOP plus its execution on the RFU is more than the time to perform the same function on the host CPU on the average.

The set ACC represents all the RFUOPs which are *accepted*, in addition to their locations on the chip. Given \mathcal{RFUOPS} and RFU dimensions W and H , the placement engine decides where to place RFUOPs.

$$ACC = \{(r_i, x_i, y_i) \mid \begin{array}{l} r_i \in \mathcal{RFUOPS}, \\ x_i \geq 0, x_i + w_i < W \\ y_i \geq 0, y_i + h_i < H \end{array}\} \quad (1)$$

where (x_i, y_i) is the coordinate on the RFU where RFUOP r_i is placed. Obviously, the conditions

$$\begin{aligned} W &\geq w_i, \quad \forall i = 1 \dots n \\ H &\geq h_i, \quad \forall i = 1 \dots n \end{aligned}$$

must be met for all the RFUOPs. Note that the cardinality of ACC set could be equal to that of \mathcal{RFUOPS} . Also, it is important to note that the placements in ACC do not allow modules to be placed out of chip boundary (See Equation 1).

The placement of RFUOPs on the RFU can be modeled as a three dimensional floorplanning problem. In a 3-D floorplanning, we have a box whose base is a rectangle with the same dimensions as the RFU ($W \times H$) and its height is the time axis (See Figure 3-a). RFUOPs are also modeled as 3-D boxes (we use $box(r_i)$ to refer to the corresponding box of the RFUOP r_i). The base of the box corresponding to RFUOP r_i is a $w_i \times h_i$ rectangle and its height is the time-span the RFUOP resides on the RFU, i.e., $(e_i - s_i)$. So, the end points of the diagonal of $box(r_i)$ have coordinates (x_i, y_i, s_i) and $(x_i + w_i - 1, y_i + h_i - 1, e_i - 1)$.

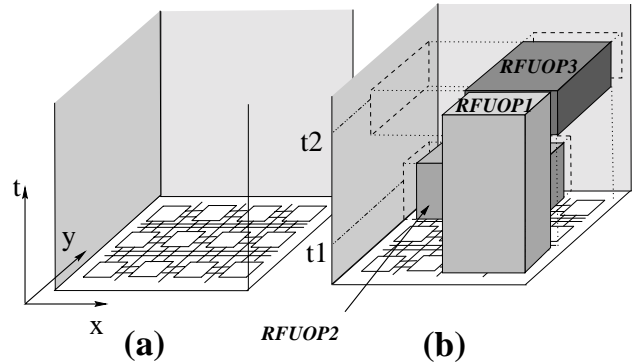


Figure 3: (a) The floorplanning box (b) A 3-D floorplan

Horizontal cuts with the floorplanning box correspond to RFU configurations at different points in time. For example, the cut $t = t_1$ in Figure 3 corresponds to Figure 1-b and the cut $t = t_2$ corresponds to Figure 1-c. Boxes corresponding to RFUOPs cannot be placed at any arbitrary point in the RFU box. The base of the RFUOP should be placed on the cut plane corresponding to $t = s_i$. However, the base can slide on the cut plane as long as it does not cross the chip boundary or other RFUOP boxes.

The penalty in rejecting an RFU operation depends both on the complexity of the operation (we assume the complexity to be linearly proportional to the size

of the module implementing the RFUOP) times number of cycles the RFUOP was supposed to take on the RFU. The number of RFU cycles could be an indication of how many times (for example in a loop) the RFUOP is supposed to be executed. We can formulate the penalty of rejecting an RFUOP r_i as $penalty(r_i)$ defined as:

$$\begin{aligned} penalty(r_i) &= w_i \times h_i \times (e_i - s_i) \\ &= volume(box(t_i)) \end{aligned} \quad (2)$$

The penalty of a placement $P \in ACC$ is defined as the sum of penalties of the rejected modules:

$$Penalty(P) = \sum_{r_i \in RFUOPS \text{ and } \exists (r_i, x, y) \in P} penalty(r_i) \quad (3)$$

The overlap of a placement $P \in ACC$ is defined as the total overlapping volume of all the RFUOP boxes:

$$Overlap(P) = \sum_{(r_i, x_i, y_i), (r_j, x_j, y_j) \in P} volume(box(r_i) \cap box(r_j)) \quad (4)$$

3-D floorplanning is the problem of finding the placement $P \in ACC$ with minimum $Penalty(P)$ and the additional constraint that no two RFUOP boxes overlap, i.e., $Overlap(P) = 0$.

3 3-D Floorplanner

Four different offline algorithms for the 3-D floorplanning problem are described in [15]. The four methods are briefly described below.

1. *KAMER-BF Decreasing*: In this method, we first sort the RFUOPs based on their box volumes, and eliminate $(100-X)\%$ smallest RFUOP boxes (X being a parameter. We tried $X = 5, 10, \dots$). Then keeping the same temporal order as the original input, give the remaining RFUOPs (largest $X\%$ modules) as the input to our best online algorithm. For a description of the *KAMER-BF* online algorithm see [1]. The reason behind eliminating the small RFUOP boxes is that, intuitively, small modules fragment the 3-D floorplan and block larger ones (with higher volume and hence larger penalties of rejection) from being placed.
2. *Simulated Annealing (SA)*: Starting from an empty 3-D floorplan, use a simulated annealing method to accept or reject RFUOPs, trying to minimize the penalty of the 3-D placement while avoiding overlaps.

3. *Low-temperature Annealing (LTSA)*: Starting from the placement generated by *KAMER-BF Decreasing*, $X\%$ use low-temperature annealing to add/remove RFUOPs to/from ACC list. All RFUOPs are considered for placement (not only the $X\%$ largest placed by the online method). An RFUOP accepted by the online method might be rejected or displaced based on the annealing decisions.

4. *Zero-temperature Annealing (ZTSA)*: Starting from the placement generated by *KAMER-BF Decreasing*, $X\%$ use zero-temperature annealing to add as many $(100-X)\%$ smallest RFUOP boxes to the ACC list as you can, trying to monotonically decrease the penalty of the placement. In contrast to LTSA method, the RFUOP boxes placed by the online algorithm are not removed or displaced. This method is greedy and much faster than LTSA.

Another more effective, faster method called Best Fit Offline Placement (BFOP) can be added to the above list. This method is not annealing based, so it completes quickly, as compared with the three annealing algorithms above.

In this method, we sort the RFUOPs according to their volume. Then, we place the modules (largest to smallest) in areas of the floorplan with the best available space. If a position can accommodate the module, i.e. adding the RFUOP will result in $Overlap(P) = 0$, then the algorithm will consider placement at that position. In addition, the position must be an upper-left or lower-right corner. Intuitively, an upper-left corner can be defined as any position (x_i, y_i) such that there is a placed RFUOP at $(x_i - 1, y_i)$ and $(x_i, y_i - 1)$. Likewise, an lower-right corner (x_i, y_i) has a placed module at $(x_i + 1, y_i)$ and $(x_i, y_i + 1)$.

By iterating through the floors between the times s_r and e_r (where s_r and e_r are the start and end times of the RFUOP considered for placement), we find all the possible corners for placement. The corners have an associated rectangular area. The RFUOP is placed in the corner so that the leftover free area is minimized. The corner chosen has the smallest remaining room after the RFUOP is placed. This method is similar to the 2-D online placement in *KAMER BF Decreasing*; only here, the RFUOP must fit in the three dimensional area whereas [15] (*KAMER BF Decreasing*) considers only the one time slice (2-D placement). The problem is similar to the well studied 2-D bin-packing which is an extension of the classical one-dimensional bin-packing.

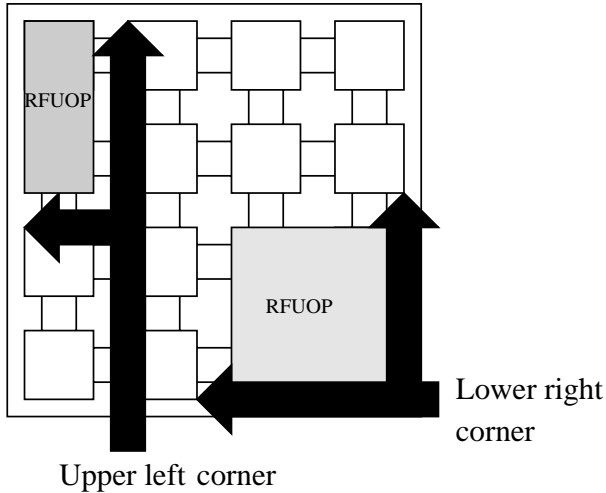


Figure 4: Example of ULC and LRC

4 Templates

Every RFUOP can be thought as a template. Simply stated, a template is the shape of the operation as it is placed on the reconfigurable unit. A RFUOP can have many templates each of which may give different benefits (area/power/delay/etc.). A hard template has only one shape, therefore the RFUOP must be placed as that shape. On the other end of the spectrum, a soft template can assume any shape. Obviously, a soft template would yield a fully packed reconfigurable unit. Unfortunately, it takes a tremendous amount of time to reshape an RFUOP since you must physically redesign the operation to fit the given area and shape.

In order to gain the flexibility of a soft template and not sacrifice the quick placement time of a hard template, we introduce the idea of a firm template. A firm template can be hierarchically broken into smaller templates. Associated with each RFUOP we have a set of pre-designed templates, where each template specifies the RFUOP at the routing level. Templates exist for commonly used operations (like adders and multipliers). An operations without a template will have to be custom mapped to the RFU, i.e., placed as a hard template.

The firm template gives you more flexibility in placing the RFUOP. For example, the dimensions of the RFUOP can be mutated to better fit the free area of the reconfigurable unit. The firm template allows splitting the RFUOP into several pieces. By splitting the RFUOP into multiple pieces, the floorplan will become less fragmented since the small unused areas are more likely to be filled. This keeps the larger areas free

and will increase the number of RFUOPs that can be placed.

While a firm template is much simpler than a soft template, it still adds complexity to the original hard template. Firstly, a split template has connections between the individual pieces. Also, the RFUOP needs a regular internal structure (e.g., a ripple carry adder is composed of many full adders). Furthermore, we assume there is a fast template routing method which will handle the necessary connections between split pieces.

When splitting a RFUOP into x pieces, the operation can be divided in a variety of different ways. A RFUOP with dimensions (w_i, h_i) can be divided such that there are x pieces with dimensions $(w_i/x, h_i)$ or x pieces with dimensions $(w_i, h_i/x)$. We will call the former a width split and the latter a height split. A best split occurs when you divide an operation either by width or height, depending on their relative lengths. If the $w_i \geq h_i$, best split will perform a width split. Else, best split will perform a height split. Another method called worst split, does the exact opposite of best split (doing a height split if $w_i \geq h_i$ and a width split otherwise). Intuitively, best split will place more modules since the split modules have a similar width and height. Worst split will create x modules that are horizontally or vertically narrow. It is more likely that there are x free areas for modules with a moderate width and height (best split) as opposed to x free areas with a large width or height (worst split).

5 Experimental Results

We use the model described in Section 2 for our insert/delete events. We generated different data sets containing the invocation of the RFUOPs. Each data set is a sequence of insertion and deletion of RFUOPs sorted by the time they occur. The events are uniformly distributed on the timeline with average density of D RFUOPs on the chip at any given time, D being a parameter of the input file. We have simulated the running of a program on the reconfigurable computing system by placing as many RFUOP boxes on the 3-D floorplan as we can. The modules which we cannot place on the RFU-time volume are rejected.

The data files are called Xnnnn (see Table 1) where the variable 'X' is the class of RFUOP module width/height distributions and 'nnnn' is number of insertion events (we have done experiments with 'nnnn' being 50, 100, 200, 1024 and 2048).

The penalty reported in the following tables is the

Data class	Min len	Max len	Avg len	D	Chip Size	Distribution
Tiny	3	30	16.5	5	50×50	Uniform
Small	3	30	16.5	10	70×70	Uniform
A	3	30	16.5	30	100×100	Uniform

Table 1: Description of different data classes. D is the density (average number of RFUOPs in the system at any time-slice).

Data file	KAMER BFD	LTSA	ZTSA	BFOP
Tiny50	213153	148975	149194	125556
Tiny100	307879	225603	261549	201055
Small100	508923	287153	486376	350969
Small200	612623	359980	571716	313146
A100	456627	213036	282587	180679

Table 2: Comparison of 3-D placement costs same as Equation 3 (sum of box volumes of rejected RFUOPs).

Table 2 shows the superiority of BFOP to the 3-D placement algorithms in [15], also described in Section 3. BFOP outperforms the other 3-D algorithms in almost every case. For this reason, we will use the results from BFOP as a base case for our firm template analysis.

Table 3 shows the results of splitting the module into x pieces (where x varies from 1 to 6). Split x pieces is the penalty or the area of the rejected RFUOPs. Ratio is a comparison between the current split number and the results when no splitting is allowed ($x = 0$). We use BFOP in order to place the RFUOPs. The RFUOPs are split only if placement of the unsplit module is unsuccessful. Of course, every portion of the split template must be placed or the RFUOP is rejected. We assume that all routing between split templates can be done in a routing channel (as is typical of most reconfigurable computing systems). The results reported use the best split heuristic. Experiments using width split, height split and worst split were not reported since the best split cost is superior. As you can see, increasing the number of pieces always decreases the penalty. But the percentage cost decrease lessens the more you split the module. There is an average increase of 23% from no splits to breaking the module into 2 parts. Compare this to the 3% decrease in penalty when going from 5 pieces to 6 pieces. In the worst case, the split x pieces modules will be placed in the same position as split $x-1$ pieces. Keep in mind that increased splitting adds more routing that may not fit in the routing channels. Also, splitting complicates the management of RFUOPs. Thus, it is

sensible to split the operations into two or three pieces.

Table 4 shows the results of BFOP when template can assume multiple shapes. The rotation column gives the results when the rotated module is allowed for placement. BFOP will place the module with its original dimensions or with its rotated dimensions depending on which “fits” the best. The median column shows the results when BFOP considers the original and median dimensions. The median dimensions transform the RFUOP into a square while keeping the same area. The median and rotate column gives the results when BFOP considers the original, median and rotated dimensions. The results show that much improvement can be gained from rotation. On the other hand, the results from the median dimensions give little to no improvement. A possible explanation for the poor results may be due to early fragmentation of the floor which blocks the smaller modules from being placed. The ratio column is always a comparison with BFOP without splitting, only considering the original dimensions.

6 Conclusion and Future Work

We summarized the results of previous work on floorplanning for reconfigurable systems and showed why it is important to deal with both online and offline placement algorithms. We introduced the ideas of a firm template and showed its benefits in offline floorplanning algorithms.

Extensive studies should be done to find realistic benchmarks for reconfigurable computing environments. These benchmarks should address the distribution of module dimensions for RFUOPs, pattern of invocation, penalty of rejection and performance gains when performing RFUOPs.

Routing of the split RFUOP needs to be considered in more detail. A router for these split modules for both the routing channel and through the cells of the floor itself should be studied.

Also, a study of firm templates in online floorplanning algorithms should be addressed. The firm templates give the possibility of decreasing the penalty of an online algorithm since more modules can be placed. This leads to increased performance. But, placing the firm templates increases the run time of the online algorithm. This cost/benefit tradeoff needs further attention.

Data Set	BFOP	Split 2 pieces	Ratio
Tiny50	125556	94876	75.56%
Tiny100	201055	177997	88.53%
Small100	350969	262174	74.70%
Small200	313146	214200	68.40%
Small1024	2488566	1806956	72.61%
A100	180679	142356	78.79%
A1024	7388012	6220709	84.20%
avg	n/a	n/a	77.54%

Data Set	Split 3 pieces	Ratio	Split 4 pieces	Ratio
Tiny50	82836	65.98%	82836	65.98%
Tiny100	116629	58.01%	147920	73.57%
Small100	249245	71.02%	246569	70.25%
Small200	181341	57.91%	186105	59.43%
Small1024	1764011	70.88%	1581278	63.54%
A100	144800	80.14%	109772	60.76%
A2048	5598256	75.77%	5333699	72.19%
avg	n/a	68.53%	n/a	66.53%

Data Set	Split 5 pieces	Ratio	Split 6 pieces	Ratio
Tiny50	82836	74.81%	84252	67.10%
Tiny100	122606	64.68%	134807	67.05%
Small100	224664	76.44%	244141	69.56%
Small200	161250	67.29%	229967	73.44%
Small1024	1477481	72.51%	1751771	70.39%
A100	104756	78.70%	160230	88.68%
A2048	5072298	78.60%	5591227	75.68%
avg	n/a	59.97%	n/a	56.75%

Table 3: The Split x pieces columns show the penalties for different data sets using BFOP while splitting the RFUOPs into x pieces. Ratio is a comparison when splitting is not allowed (the BFOP column)

Data Set	BFOP	Rotation	Ratio
Tiny50	125556	82576	65.77%
Tiny100	201055	128125	63.73%
Small100	350969	246659	70.28%
Small200	313146	231189	73.83%
Small1024	2488566	1956808	78.63%
A100	180679	156024	86.35%
A1024	7388012	6199460	83.91%
avg	n/a	n/a	74.64%

Data Set	Median	Ratio	Median/rotation	Ratio
Tiny50	103630	82.64%	78988	62.91%
Tiny100	181922	90.48%	108081	53.76%
Small100	290633	82.81%	234310	66.76%
Small200	312254	99.72%	201418	64.32%
Small1024	2509154	100.83%	2025975	81.41%
A100	197427	109.27%	180110	99.69%
A2048	7512160	101.58%	6143759	83.16%
avg	n/a	95.33%	n/a	73.14%

Table 4: The Median, Rotation, and Median/Rotation columns show the penalties for different data sets using BFOP while considering multiple dimensions. Ratio is a comparison with BFOP allowing only the original dimensions.

References

- [1] K. Bazargan and M. Sarrafzadeh. "Fast Online Placement for Reconfigurable Computing Systems". To appear in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.
- [2] G. Brebner. "The Swappable Logic Unit: a Paradigm for Virtual Hardware". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, 1997.
- [3] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. Wit. "A Dynamic Reconfiguration Run-Time System". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 66–75, 1998.
- [4] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. "Fast Module Mapping and Placement for Datapaths in FPGAs". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages –, February 1998.
- [5] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. "Splash: A Reconfigurable Linear Logic Array". In *International Conference on Parallel Processing*, pages 526–532, 1990.
- [6] S. Hauck. "Configuration Prefetch for Single Context Reconfigurable Coprocessors". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 65–74, February 1998.
- [7] S. Hauck. "The Roles of FPGAs in Reprogrammable Systems". *Proceedings of the IEEE*, 86(4):615–638, April 1998.
- [8] S. Hauck, Z. Li, and E. J. Schwabe. "Configuration Compression for the Xilinx XC6200 FPGA". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 138–146, 1998.
- [9] C. Iseli and E. Sanchez. "Spyder: A Reconfigurable VLIW Processor using FPGAs". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 17–24, 1993.
- [10] H. Krupnova, C. Rabedaoro, and G. Saucier. "Synthesis and Floorplanning for Large Hierarchical FPGAs". In *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pages –, February 1997.
- [11] H. Liu and D. Wong. "Circuit Partitioning for Dynamically Reconfigurable FPGAs". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages –, 1999.
- [12] J. Shi and D. Bhatia. "Performance Driven Floorplanning for FPGA Based Designs". In *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 112–118, February 1997.
- [13] W. J. Sun and C. Sechen. "Efficient and Effective Placement for Very Large Circuits". *IEEE Transactions on Computer Aided Design*, 14(3):349–359, March 1995.
- [14] M. J. Wirthlin and B. L. Hutchings. "A Dynamic Instruction Set Computer". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, 1995.
- [15] K. Bazargan, R. Kastner and M. Sarrafzadeh. "3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems" *Proceedings of IEEE Workshop on Rapid System Prototyping*, pages 38–43, 1999.