Design Automation for Embedded Systems, 5, 329–338 (2000)
© 2000 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems

KIARASH BAZARGAN kiarash@ece.nwu.edu
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118

RYAN KASTNER kastner@ece.nwu.edu
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118

MAJID SARRAFZADEH majid@ece.nwu.edu
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118

Abstract. The advances in the programmable hardware have 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). In reconfigurable systems, we are interested both in online placement, where the arrival time of tasks is determined at runtime and is not known *a priori*, and offline in which the schedule is known at compile time. In the case of offline placement, we are willing to spend more time during compile time to find a compact floorplan for the RFU modules and utilize the RFU area more efficiently. In this paper we present offline placement algorithms based on simulated annealing and greedy methods and show the superiority of their placements over the ones generated by an online algorithm.

Keywords: Reconfigurable computing, floorplanning, simulated annealing.

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 200× 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

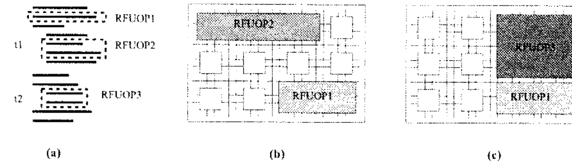

Figure 1. (a) The running code. (b) RFU configuration at time t1. (c) RFU configuration at a later time t2.

is swapped out of the chip and RFUOP3 is loaded. RFUOP1 is still on the chip and may be called later in the program.

Unfortunately, rather long delays in reprogramming RFUs keep us from achieving very high speedups in general purpose computing. Wirthlin and Hutchings [14] report an overall speedup of 23×, while the speedup could behave 80× 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. The importance of the online version is that 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 kernels. 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 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], [12], [11]

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 online algorithm and as a

baseline for assessing the quality of online methods. We propose simulated annealing as well as greedy offline algorithms for placement of the modules on RFU, and show the effectiveness of the proposed methods by comparing their placements with some online algorithms (see [1]).

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. Experimental results are shown in Section 4. Section 5 contains 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).

Unlike [3], our model allows no rotation or flipping of the modules. They should be placed on the RFU as they are represented in the library. Furthermore, we assume there is only one representation for each RFUOP in the library. There is no connection between RFUOPs. The data to be processed by the RFUOP is loaded onto chip registers before the RFUOP starts execution. After the RFUOP is done, the result is read back to the CPU.

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

The set

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

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

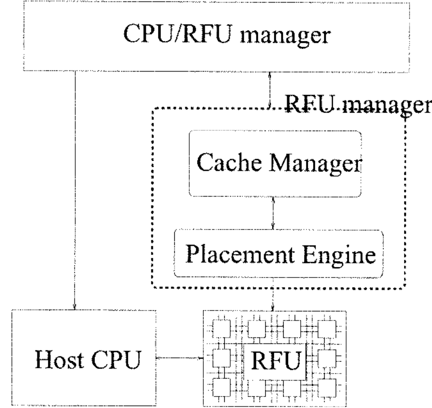

Figure 2. A sample model of a reconfigurable computing system.

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.

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 $\mathcal{ACC}$ 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.

$$\mathcal{ACC} = \{\{(r_i, x_i, y_i) \mid r_i \in \mathcal{RFUOPS}, \text{Placement}(r_i) = x_i, y_i)\}\} \tag{1}$$

where (x_i, y_i) is the coordinate on the RFU where RFUOP r_i is placed by the placement engine. Obviously, the following conditions must be met for all the RFUOPs.

$$H \geq h_i, \quad W \geq w_i, \quad \forall i = 1 \ldots n$$
$$x_i \geq 0, x_i + w_i < W$$
$$y_i \geq 0, y_i + h_i < H$$

Note that the cardinality of $\mathcal{ACC}$ set could be equal to that of $\mathcal{RFUOPS}$. Also, it is important to note that the placements in $\mathcal{ACC}$ do not allow modules to be placed out of chip

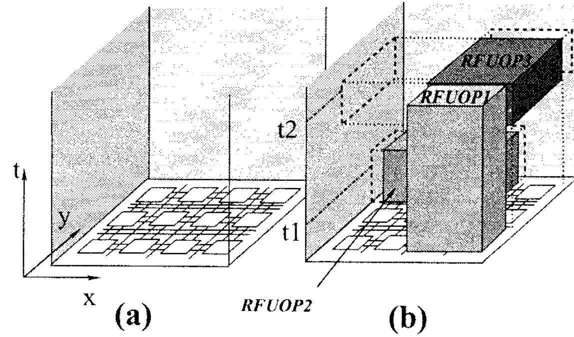

Figure 3. (a) The floorplanning box. (b) A 3-D floorplan.

boundary (See Equation 1), but some RFUOP boxes might overlap. We will deal with this issue in Section 3.

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 $(x_i + w_i - 1, y_i + h_i - 1, e_i - 1)$.

Horizontal cuts with the floorplanning box correspond to RFU configurations at different points in time. For example, the cut $t = t1$ in Figure 3 corresponds to Figure 1-b and the cut $t = t2$ 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.

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:

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

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

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

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

$$Overlap(P) = \sum_{(r_i,x_i,y_i),(r_j,x_j,y_j)\in P, i\neq j} box(r_i) \cap box(r_j) \tag{4}$$

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

3. 3-D Floorplanner

We implemented four different offline algorithms for the 3-D floorplanning problem. The four methods are listed 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 $X = 5, 10, \ldots$). 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]. We are willing to eliminate small modules, because 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.

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 $P \in \mathcal{ACC}$ 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 $P \in \mathcal{ACC}$ 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.

The annealing core is the same for SA, LTSA and ZTSA methods. Their only difference is in the starting temperature of the annealing process. The annealing core starts with an element $P_0 \in \mathcal{ACC}$ and applies three different moves to generate other placements

$P_1, P_2, \ldots$ where $P_i \in \mathcal{ACC}$ trying to minimize $penalty(P)$. The moves are:

1. *Op1: Accept RFUOP*

   — Precondition: $r_i \in \mathcal{RFUOPS}$, $(r_i, x, y) \notin P$

   — Operation:

$$P \leftarrow P \cup \{(r_i, rand(0, W - w_i), rand(0, H - h_i))\}$$

   where $rand(a, b)$ generates a random integer number in the range $(a, b - 1)$.

   — Post-condition: $Overlap(P) = 0$

   In fact, we use a more mature way than generating random coordinates to choose the location of the RFUOP box. We look for all possible empty boxes which can accomodate $box(r_i)$ (See [1] for similar methods in two dimensions) and choose one randomly.

2. *Op2: Reject RFUOP*

   — Precondition: $(r_i, x_i, y_i) \in P$

   — Operation: $P \leftarrow P \setminus \{(r_i, x_i, y_i)\}$

3. *Op3: Displace RFUOP*

   — Precondition: $(r_i, x_i, y_i) \in P$

   — Operation:

$$\begin{aligned} x_{i\,new} &\leftarrow x_i + rand(-\delta, \delta) \\ y_{i\,new} &\leftarrow y_i + rand(-\delta, \delta) \\ P &\leftarrow (P \setminus \{(r_i, x_i, y_i)\}) \cup \left\{\left(r_i, x_{i\,new}, y_{i\,new}\right)\right\} \end{aligned}$$

   — Post-condition:

$$\begin{aligned} x_{i\,new} \geq 0 \text{ and } x_{i\,new} < W - w_i \\ y_{i\,new} \geq 0 \text{ and } y_{i\,new} < H - h_i \\ Overlap(P) = 0 \end{aligned}$$

The selection of the RFUOPs to add to the $P \in \mathcal{ACC}$ list (i.e., accept) or remove from the $P \in \mathcal{ACC}$ list (i.e., reject) or displace is done randomly. In Section 5 we will discuss the effect of choosing RFUOPs with different probabilities. We have also discussed the effect of choosing the annealing operations with different probabilities.

$Penalty(P_i)$ is used as the cost for each placement. Note that we could have allowed overlaps between RFUOP boxes and try to resolve it towards the end of the annealing process. In that case, the cost would have been $Penalty(P) + \lambda(T) \times Overlap(P)$, where λ is an increasing function of annealing temperature T, to ensure that the overlap cost converges to zero at the end of the annealing process. We did perform experiments with this method, but the method which allows no overlaps to occur is faster. (The authors in [13] report that 2-D placement methods which allow/prevent overlaps generate placements of fairly equal qualities).

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

Data class	Min Len	Max Len	Avg Length	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 II. Comparison between LTSA-100 and online (KAMER-BFD). Both acceptance rate and penalties are given.

Data Set	LTSA-100 acc. rate	Online acc. rate	Ratio acc. rate	LTSA-100 penalty	Online penalty	Ratio penalty
Tiny50	70	84	83.33%	147287	213153	69.10%
Tiny100	72	83	86.75%	253566	307879	82.36%
Small100	86	84	102.38%	464049	508923	91.18%
Small200	81	89.5	90.50%	539435	612623	88.05%
Small1024	84.47	84.57	99.88%	4468662	4643786	96.23%
A100	87	89	97.75%	427761	456627	93.68%

4. 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 Cnnnn (see Table I) where ‘C’ 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 same as Equation 3 (sum of box volumes of rejected RFUOPs). The tables show the ratio of accepted RFUOPs to the total number of RFUOPs as well.

The experiments with different values of X for KAMER-BF Decreasing method showed that using $X < 93$ result in higher penalties than $X = 100$. In the cases where $X \geq 93$, slight improvements in the penalty of the placement was seen, and hence we did not report the results of these experiments. Also, pure annealing took long times (e.g., hours for Small100 data set) and hence we did not report the results of SA either. However, LTSA and ZTSA methods yielded good results.

Table II shows the ratio of accepted RFUOPs when the output of KAMER-BF Decreasing with X=100% is used as input to the low-temperature annealing (LTSA) method. The results of LTSA are compared to the online algorithm (KAMER-BFD with X=100, see [1]). In the same table, the penalties of the two methods are also shown. As can be seen, the acceptance

Table III. The LTSA-20 columns correspond to KAMER-BFD with $X = 20\%$ followed by LTSA. Online columns correspond to KAMER-BFD with $X = 100$.

Data penalty Set	LTSA-20 penalty acc. rate	Online penalty acc. rate	Ratio acc. rate	LTSA-20	Online	Ratio
Tiny50	76	84	90.48%	148975	213153	69.89%
Tiny100	82	83	98.79%	225603	307879	73.28%
Small100	81	84	96.43%	287153	508923	56.42%
Small200	85.5	89.5	95.53%	359980	612623	58.76%
A100	81	89	91.01%	213036	456627	46.65%

Table IV. The ZTSA-20 columns correspond to KAMER-BFD with $X = 20\%$ followed by ZTSA. Online columns correspond to KAMER-BFD with $X = 100$.

Data Set	ZTSA-20 acc. rate	Online acc. rate	Ratio acc. rate	ZTSA-20 penalty	Online penalty	Ratio penalty
Tiny50	74	84	88.09%	149194	213153	69.99%
Tiny100	79	83	95.18%	261549	307879	84.95%
Small100	74	84	88.09%	486376	508923	95.57%
Small200	77	89.5	86.03%	571716	612623	93.32%
A100	73	89	82.02%	282587	456627	61.88%

rate decreases in some cases but the penalty always improves. The reason is that smaller RFUOP boxes are replaced by larger ones, hence increasing number of rejected modules but decreasing the penalty. Table III is similar to Table II, but X is set to 20, instead of 100. As can be seen, the LTSA method is able to improve the online results substantially.

Table IV shows the acceptance rate and penalties for the case where KAMER-BF Decreasing with $X = 20\%$ is run first, and its placement is used as starting point for the ZTSA method. The ZTSA only accepts the RFUOPs which are not placed by the online algorithm, and hence is very fast. It can be seen that although it is a greedy method, it still can improve the results of the online method.

5. 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 devised simulated annealing and greedy placement methods for the 3-D placement of the RFUOPs and showed their effectiveness.

The effect of different representations of RFUOPs in the library should be addressed in future work. Also, 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.

Another important issue to be addressed is the effect of weighting different modules when choosing them for insertion/deleting into the active tasks. It would be interesting to observe how the result of our method changes if modules with smaller volumes are more likely to be removed from the active tasks list. The small modules probably fragment the floorplanning box and cause rejection of larger modules and hence increase the overall penalty. Also, the effect of selecting the four annealing moves (See Section 3) with different probabilities should be examined.

Acknowledgment

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

References

1. Bazargan, K., and Sarrafzadeh, M. 1999. Fast online placement for reconfigurable computing systems. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*.
2. Brebner, G. 1997. The swappable logic unit: a paradigm for virtual hardware. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* pp. 77–86.
3. Burns, J., Donlin, A., Hogg, J., Singh, S., and Wit, M. 1998. A dynamic reconfiguration run-time system. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* pp. 66–75.
4. Callahan, T. J., Chong, P., DeHon, A., and Wawrzynek, J. 1998. Fast module mapping and placement for datapaths in FPGAs. *International ACM/SIGDA Symposium on Field Programmable Gate Arrays* February.
5. Gokhale, M., Holmes, B., Kopser, A., Kunze, D., Lopresti, D., Lucas, S., Minnich, R., and Olsen, P. 1990. Splash: a reconfigurable linear logic array. *International Conference on Parallel Processing* pp. 526–532.
6. Hauck, S. 1998. Configuration prefetch for single context reconfigurable coprocessors. *International ACM/SIGDA Symposium on Field Programmable Gate Arrays* pp. 65–74, February.
7. Hauck, S. 1998. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE* 86(4): 615–638.
8. Hauck, S., Li, Z., and Schwabe, E. J. 1998. Configuration compression for the Xilinx XC6200 FPGA. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* pp. 138–146.
9. Iseli, C., and Sanchez, E. 1993. Spyder: a reconfigurable VLIW processor using FPGAs. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* pp. 17–24.
10. Krupnova, H., Rabedaoro, C., and Saucier, G. 1997. Synthesis and floorplanning for large hierarchical FPGAs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)* , February.
11. Liu, H., and Wong, D. F. 1999. Circuit partitioning for dynamically reconfigurable FPGA. *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*.
12. Shi, J., and Bhatia, Dinesh. 1997. Performance driven floorplanning for FPGA based designs. *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)* pp. 112–118, February.
13. Sun, W. J., and Sechen, C. 1995. Efficient and effective placement for very large circuits. *IEEE Transactions on Computer Aided Design* 14(3): 349–359, March.
14. Wirthlin, M. J., and Hutchings, B. L. 1995. A dynamic instruction set computer. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* pp. 99–107.