Transactions Briefs

Placement for Large-Scale Floating-Gate Field-Programable Analog Arrays

Faik Baskaya, Sasank Reddy, Sung Kyu Lim, and David V. Anderson

***Abstract*—Modern advances in reconfigurable analog technologies are allowing field-programmable analog arrays (FPAAs) to dramatically grow in size, flexibility, and usefulness. Our goal in this paper is to develop the first placement algorithm for large-scale floating-gate-based FPAAs with a focus on the minimization of the parasitic effects on interconnects under various device-related constraints. Our FPAA clustering algorithm first groups analog components into a set of clusters so that the total number of routing switches used is minimized and all IO paths are balanced in terms of routing switches used. Our FPAA placement algorithm then maps each cluster to a computational analog block (CAB) of the target FPAA while focusing on routing switch usage and balance again. Experimental results demonstrate the effectiveness of our approach.**

***Index Terms*—Analog circuit, field-programmable analog arrays (FPAAs), placement.**

I. Introduction

The ever-increasing demand in lower power systems makes analog solutions to certain signal processing tasks a promising alternative to digital processing. Although advances in field-programmable analog arrays (FPAAs) allow reconfigurable low-power analog designs on standard CMOS technology, FPAAs still have not achieved the same success as field-programmable gate arrays (FPGAs) due to the lack of computer-aided design (CAD) tools and the nonideal programming technology, which contributes a large portion of parasitics into the sensitive analog system.

This paper is focused on making our large-scale floating-gate-based FPAA technology [1] more accessible and practical. An illustration of a small analog circuit mapping using our FPAA is shown in Fig. 1. Our analog CAD tool automates the placement of analog circuit blocks on a target large-scale FPAA. Our FPAA clustering algorithm groups analog components into a set of clusters so that the total number of routing switches used is minimized and all IO paths are balanced in terms of routing switches used. Our FPAA placement algorithm maps each cluster to a computational analog block (CAB) in a target FPAA while focusing on routing switch usage and balance again. Experimental results demonstrate the effectiveness of our approach.

A floating-gate element is a polysilicon layer with no contacts to other layers; this polysilicon layer can be the gate of a MOSFET and can be capacitively connected to other layers and can maintain a permanent charge as an analog memory cell. The charge on the floating gate is modified through a combination of hot-electron injection and electron tunneling [2]. Floating-gate transistors function in two ways in the FPAA: as switches that connect device pins or as configuration devices for analog components in the CAB. Using floating-gate transistors as switches has advantages over using a standard pFET or a transmission gate. The switch resistance exhibits more linearity and consistency over the operating voltage ranges [3] and can be controlled by the injection amount, allowing building them with small aspect ratios and, therefore, smaller sizes and larger arrays. Floating-gate switches can also serve as transistors with fixed gate voltages acting as anything from resistors and spreading elements to current sources.

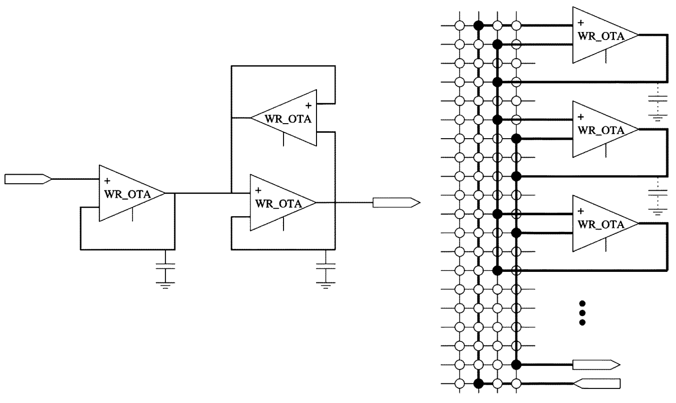

Fig. 1. Analog circuit and its mapping onto our floating-gate-based FPAA.

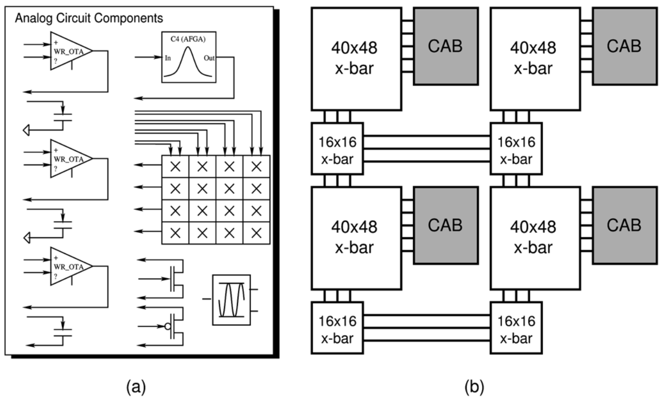

Fig. 2. (a) CAB for an FPAA based on floating-gate devices. (b) Overall block diagram for a large-scale FPAA.

The computational logic in the FPAA is organized in a compact CAB that consists of op-amps, transistors, multiplier, capacitors, edge detectors, and filters as illustrated in Fig. 2. CABs are tiled across the chip in a regular mesh-type architecture with busses and local interconnects in between similar to the tiling in simple FPGA architectures. The major parasitic effects on FPAA chips are due to parasitic resistance and capacitance of the routing switches on FPAA interconnects. Unlike digital circuits, these parasitics have a cumulative impact on the performance of analog circuits. Therefore, the primary objective during physical synthesis is to minimize the number of switches connected to a signal path and to balance the different signal paths. Since the routing metals have been prebuilt and the only control given is on the switches,

Manuscript received April 1, 2005; revised December 26, 2005. This work was supported in part by the National Science Foundation under Contract CNS-0411149.

F. Baskaya, S. K. Lim, and D. V. Anderson are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: baskaya@ece.gatech.edu).

S. Reddy is with the Department of Electrical Engineering, University of California, Los Angeles, CA 90095 USA.

Digital Object Identifier 10.1109/TVLSI.2006.878477

the mapping algorithm will not be affected from the changes in technology except for the necessity of updating the actual switch and wire parasitics.

II. Existing Work

There currently exist several CAD efforts for FPAAs in the literature [4], [5]. However, these works focus only on small-scale, switch-capacitor-based designs. On the other hand, our floating-gate-based FPAAs contain up to 144 (12×12) complex CABs, thereby necessitating a more scalable approach to handle the complexity. In addition, the device and interconnect constraints in our floating-gate-based FPAA are radically different from the switch-capacitor-based FPAA. Significant work has been done on logical and physical synthesis for lookup-table (LUT)-based FPGAs during the last 20 years [6]–[8].

The device and interconnect constraints in floating-gate-based FPAA make traditional FPGA algorithms unapplicable. During the placement phase, FPGA architectures often use simulated annealing or genetic algorithms to achieve an optimal solution. Although these same ideas can be applied to FPAA implementations, the cost functions in these algorithms vary. Typically, FPGA and, in general, most digital interfaces place higher priority on reducing wire length, net density, and overall delay. On the other hand, FPAA and analog architectures typically focus on reducing loading effects and distortions of signals.

FPGAs traditionally have a very simple interconnect network architecture, where the vertical and horizontal channels have the same number of routing tracks and propagate the same signal down a pipeline. Each horizontal and vertical wire is segmented and are shared among several interconnects. This varies quite significantly from our FPAA, where the interconnect wires are not segmented and a single interconnect occupies the entire vertical/horizontal highway. The basic logic element used in FPGAs is a combination of programmable LUTs with universal functionality and flip-flops in general, whereas a CAB in an FPAA consists of many more various programmable analog components with distinct functionality. In addition, there are no sequential elements in an FPAA. Thus, behavioral as well as physical synthesis needs entirely different approaches to handle new cost functions under new types of device constraints. Thus, it is fairly difficult to directly associate automation algorithms between FPGAs and FPAAs.

III. FPAA Interconnect Analysis

A. Interconnect Modeling

There are three types of routing switch boxes in floating-gate-based FPAA: local, vertical, and horizontal crossbars that establish connections between components in the same CAB, CABs in the same column, and CABs in different columns, respectively. The routing switches in local, vertical, and horizontal crossbars are, respectively, called local, vertical, and horizontal switches. Three types of wires are connected through these switches to each crossbar (see Fig. 3).

- Type 1: Intra-CAB (local) wires connect components in the same CAB using the switches in local crossbar. The parasitics of these wires are minimal.
- Type 2: Inter-CAB/intra-column (vertical global) wires connect components from different CABs located in the same column and extend all the way from top to bottom.
- Type 3: Inter-column (horizontal global) wires connect components from different CABs located in different columns and extend all the way from left to right. The parasitics of these wires are maximal.

Local (type 1) wires are alternatively called *i-nets* in this paper, whereas global (types 2 and 3) wires are called *x-nets*. Since global wires span the whole FPAA vertically or horizontally, each global

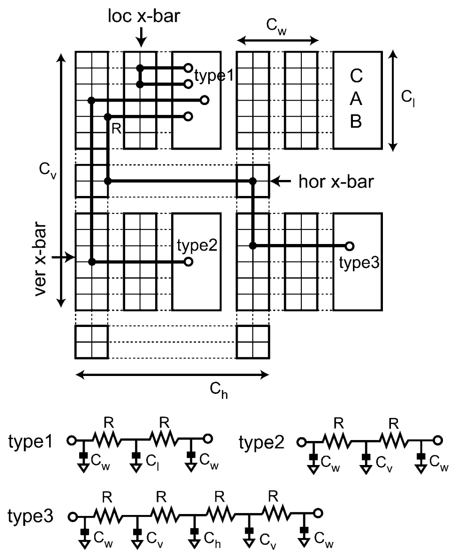

Fig. 3. Routing resource for a 2×2 FPAA that consists of local, vertical, and horizontal crossbars. Three types of interconnects (1: intra-CAB; 2: inter-CAB/intra-column; and 3: inter-column) are also shown.

wire can only accommodate a single interconnect. In addition, once a net is occupying a wire, the sum of the resistance and capacitance of *all* routing switches in this wire contributes to the parasitics of the interconnect. We model the resistance of switches that are turned on $(= R)$ and sum of all of the off switch capacitances on a horizontal local wire $(= C_w)$, vertical local wire $(= C_l)$, horizontal global wire $(= C_h)$, and vertical global wire $(= C_l)$. We assume that $C_w < C_v$, $C_w < C_h$, and $C_l < C_v$. In case the interconnect contains more than two components (= multi-pin net), each source-to-sink connection can be individually modeled. Zero-value time constant analysis [9] on the interconnect type 1 in Fig. 3 yields $\omega \approx 1/R(2C_w + C_l)$ for the dominant pole only. The equation shows that each switch contributes a pole when added into the circuit path and has a negative impact on the bandwidth.

B. Path Delay Balance

In digital circuits, unequal delay in parallel signal paths results in reduced clock speed. In analog circuits, however, such a situation threatens signal integrity when signals from unbalanced paths (= unequal numbered switches on these paths) are combined together. Therefore, path length balance is critical in FPAA physical synthesis. Our strategy is to identify the groups of related paths that need to be balanced based on their initial path lengths, i.e., before the routing switches are inserted. These groups are determined using the k-means clustering method [10]. We then attempt to insert an equal amount of routing switches along the paths in the same group as much as possible to balance their delay.

IV. FPAA Clustering Algorithm

A. Problem Formulation

The goal of FPAA clustering is to pack the analog circuit components that are closely connected to each other into the same CABs. This process is constrained by device, internal net (i-net), and external net (x-net) limitations. The objective is to maximize the device and i-net (= local wire) utilization while minimizing the x-net (= vertical and horizontal global wires) usage in order to make room for the subsequent placement phase. Balancing the delay of parallel paths by assigning fewer switches to the longer paths is another objective. As mentioned

TABLE I
BENCHMARK CIRCUITS

ckt	CAB	#cells	#nets	ckt	CAB	#cells	#nets
c1	fpaa1	10	16	c11	fpaa2	217	245
c3	fpaa1	20	27	c13	fpaa2	326	356
c5	fpaa2	44	55	c15	fpaa3	395	482
c7	fpaa2	110	147	c17	fpaa3	438	484
c9	fpaa2	118	143	c19	fpaa3	534	602

TABLE II
FPAA ARCHITECTURES

comp	fpaa1	fpaa2	fpaa3
dimension	4×4	8×8	12×12
cab0	4	16	36
cab1	12	48	108
local wires	16×10	64×10	144×10
vertical global wires	4×6	8×15	12×33
horizontal global wires	4×8	8×8	12×8

in Section III-A, each wire is not segmented and can only be used by a single net (either two-pin or multiple-pin). Thus, care must be taken to balance the usage of these wires to prevent unroutable solutions.

B. Clustering Algorithm

Our constructive CAB clustering algorithm consists of three steps: 1) preclustering of the user defined groups; 2) cell ordering for clustering priority; and 3) CAB ranking and selection. In a nutshell, we visit the cells in a certain order and search for the best possible CAB to cluster with while monitoring various constraints. We temporarily allow x-net constraint violation and correct it during the later stage of the clustering process.

First, the cells specified by the user constraints are clustered into groups that have priority with respect to number of cells within. Then comes the vector matrix multiplier (VM) and, finally, the remaining cells in the order determined as follows. In *net-driven ordering*, the cells are ordered according to the modified hyper edge coarsening (MHEC) scheme [11]. MHEC is used in circuit partitioning for cutsize minimization. In this scheme, we visit the nets (hyperedges) in ascending order of their sizes, where the size of a net is the number of cells it connects. From each net we visit, we pick a random cell. The motivation is to break as few nets as possible by focusing on smaller nets first, so that the total number of intercluster connections is naturally minimized. In *path-driven ordering*, we perform static timing analysis to compute the timing slack values for all cells to be ordered. We then sort the cells in ascending order of timing slack values. This method encourages timing critical nodes to be clustered together during the early stage of clustering, thereby reducing the longest path delay (= maximum number of switches used among all paths), which has been shown to have a nontrivial impact on path delay balancing. In our hybrid *net/path-driven ordering* approach, we use MHEC to visit nets and order the cells in each net using timing slack values. These approaches are compared against a random ordering scheme in Section VI.

For each cell in the sorted order, every CAB is scanned for availability and then ranked. Ranking is done based on the improvement of occupancy of the CAB, increase in the number of i-nets, and decrease in the number of x-nets when the given cell is assigned to the CAB of interest. Then, the CAB with the highest rank is selected for merging. In case there exists no feasible CAB for a given cell, we allow the x-net constraint to be violated and attempt to fix it using our x-net_reduction method. The objective of this algorithm is to reduce the number of intercluster connection (= x-net) for a given cluster to an acceptable level L, where L could be equal to the x-net limit or even lower. For a given cluster C, a neighboring cell n is called *feasible* if: 1) n is not clustered yet; 2) there is a net that connects any cell in C and n; and 3) merging C and n does not violate the device and i-net constraints. For each feasible neighbor n for a given cluster C, we compute pkey that denotes the number of nets incident on c that do not connect to C. Then, the neighbor with the minimum pkey is selected for merging with C. If there are several neighbors with the same minimum pkey value, then we break the tie using another value named skey. The skey of a cell n is the number of nets that connect C and n together. In case there exists no more feasible neighbor or the x-net is not reduced below the threshold value, we conclude that x-net reduction is impossible for this cluster. We use a binary-heap-based priority queue for neighbor management. Thus, this algorithm runs in $O(n \log n)$.

The CAB rank equation for a given CAB C is computed as follows:

$$\alpha \times (\Delta\text{inets} - \Delta\text{xnets} - \#\text{cut}) + \beta \times \frac{\#\text{occ}}{\#\text{all}} \quad (1)$$

where #occ (#all) is the number of occupied (all) slots in the CAB, α and β are weighting constants, $\Delta\text{inets}(\Delta\text{xnets})$ is the change in number of i-nets (x-nets), and #cut is the number of nets added into the cutset by choosing this CAB for merging. The complexity of the algorithm is $O(m \times n)$, where m is the total number of CABs in the target FPAA and n is the total number of components in the given analog circuit.

V. FPAA PLACEMENT ALGORITHM

A. Problem Formulation

FPAA placement is a process of performing one-to-one mapping between the set of CAB clusters and the physical CABs in the given FPAA architecture. The main objective of the FPAA placement is to reduce the parasitic effects of the inter-CAB interconnects and switches while maintaining the path delay balance. We accomplish this goal by reducing the total number of vertical and horizontal global switches used, which in turn requires the minimization of the number of columns that each x-net spans. In case an x-net spans CAB clusters in $m > 1$ distinct CAB columns, we need to use at least m vertical global wires and one horizontal global wire (= type-3 wires in Fig. 3). Otherwise, we use only a single vertical global wire (= type-2 wires in Fig. 3). In addition, the balance in terms of the total number of global switches used in all paths is another important goal. It is also important to minimize the longest path delay to reduce the overall latency of the mapped solution. Two types of constraints are imposed during FPAA placement: CAB type and x-net constraints.

B. Placement Algorithm

Our FPAA placement consists of two steps: constructive placement and stochastic refinement. We construct an initial CAB placement solution while reducing inter-CAB-column connections under various device-related constraints. This solution is then iteratively refined by a series of moves and swaps of CABs during our simulated annealing-based refinement.[1]

The goal of FPAA constructive placement is to assign CAB clusters to CAB columns so that the number of intercolumn connections is minimized. In our simple heuristic, the CAB clusters are arranged in descending order of the number of their x-nets. The CAB columns in the target FPAA device are arranged in ascending order of the number

[1]A relatively small number of CABs that need to be placed (2–60 CABs) motivated us to attempt an ILP-based approach for CAB placement. We observed, however, that, due to the complex x-net constraint, the runtime involved was prohibitive.

TABLE III
FPAA CLUSTERING RESULTS, WHERE % UTILIZATION OF CABs, INTERNAL NETS, AND EXTERNAL NETS IS SHOWN. WE ALSO REPORT THE MAXIMUM LONGEST PATH DELAY (xlp). THE TOTAL RUNTIME FOR ALL 20 CIRCUITS IN SECONDS IS REPORTED FOR EACH ALGORITHM

	random				net-driven				net/path-driven				path-driven			
ckt	cab	i-net	x-net	xlp	cab	i-net	x-net	xlp	cab	i-net	x-net	xlp	cab	i-net	x-net	xlp
c1	33.33	23.33	30.00	13	33.33	33.33	20.00	6	33.33	33.33	20.00	6	33.33	33.33	20.00	6
c3	28.57	14.29	24.29	43	40.00	32.00	22.00	29	40.00	32.00	22.00	29	33.33	21.67	23.33	30
c5	61.82	27.14	51.43	61	54.09	28.75	40.00	69	61.82	30.00	48.57	69	54.20	26.25	42.50	61
c7	56.36	27.89	49.47	68	56.41	34.74	42.63	60	56.41	34.74	42.63	60	62.57	46.47	40.00	46
c9	61.44	22.11	53.16	99	61.53	34.21	41.05	78	61.44	33.16	42.11	78	58.32	27.00	44.50	71
c11	67.59	24.38	52.19	76	65.62	31.82	42.42	85	65.62	31.82	42.42	85	65.62	28.48	45.76	78
c13	77.45	30.95	53.81	39	85.67	54.21	39.47	60	85.67	55.00	38.68	67	85.67	51.32	42.37	67
c15	64.79	16.83	63.50	342	67.01	27.93	55.17	296	64.80	26.33	54.00	263	64.79	25.83	54.50	262
c17	83.92	11.15	81.92	34	83.97	40.58	52.50	34	83.95	39.23	53.85	20	83.92	33.08	60.00	34
c19	73.88	10.14	73.47	26	73.94	37.64	45.97	28	73.88	34.72	48.89	35	73.89	35.56	48.06	19
ave	65.19	21.54	55.41	66.7	67.36	37.11	42.29	60.4	67.28	36.89	42.41	58.05	66.32	34.31	43.87	56.8
time (s)	404				412				413				409			

TABLE IV
SIMULATED-ANNEALING-BASED FPAA PLACEMENT RESULTS WITH VARIOUS COST FUNCTIONS. WE REPORT THE TOTAL USAGE OF HORIZONTAL GLOBAL SWITCHES (hgsw), MAXIMUM LONGEST PATH DELAY (xlp), AND PATH-BALANCING MEAN-SQUARED ERROR (mse)

	hgsw-only			mse-only			xlp-only			hgsw + mse + xlp		
ckt	hgsw	mse	xlp	hgsw	mse	xlp	hgsw	mse	xlp	hgsw	mse	xlp
c1	2	0.25	6	2	0.25	6	2	0.25	6	2	0.25	6
c3	14	1014.00	125	14	1014	125	14	1014.00	125	14	1014.00	125
c5	63	1040.22	325	63	1040.22	325	63	1040.22	325	63	1040.22	325
c7	116	1358.64	252	123	19.76	220	124	339.76	206	125	19.76	220
c9	124	4692.25	270	132	1089	231	137	1089.00	231	133	1089.00	231
c11	180	36.75	341	227	0	309	211	908.75	174	194	0.00	238
c13	224	338.00	259	272	10.89	291	264	288.22	99	231	10.89	163
c15	507	1477.75	1142	602	8.33	1165	558	297.17	537	503	15.08	679
c17	419	219.17	84	527	0.5	84	498	85.83	52	440	0.50	84
c19	535	1152.89	163	623	32.67	99	633	32.67	35	548	32.67	99
ave	216.80	813.44	238.40	258.8	183.76	220.55	252.00	303.42	147.45	222.40	185.42	172.15
time (s)	4267			4026			4110			4736		

of x-nets that are used. Since each CAB column contains only a few CABs, it is usually not possible to assign all CABs in the same x-net. In addition, the x-net constraint quickly becomes an issue during our constructive placement. Thus, we focus more on x-net management during our constructive placement. Since the CAB clusters can be of different types, the algorithm considers the type constraint as well. The CAB clusters are safely placed in columns if the type restriction and the x-net limit are satisfied. If one of the restrictions fails, than the next column is considered. This process repeats until a feasible column is found for a given CAB. In case there exists no feasible column for a given CAB, we allow x-net violation and correct it during our simulated-annealing (SA)-based refinement. This violation becomes a penalty term in our SA cost function and is minimized along with other cost factors. The row assignment of CABs in each CAB column is done randomly at the end of column assignment.

SA is a widely used and well-developed stochastic placement refinement method and is employed as the last step of our hierarchical placement tool. Our SA-based refinement is based on the following cost function:

$$\mathrm{cost} = \alpha \times \mathrm{hgsw} + \beta \times \mathrm{mse} + \gamma \times \mathrm{xpl} \quad (2)$$

where hgsw, mse, and xpl, respectively, denote the total number of horizontal global switches used, the mean square error introduced in Section III-B, and the maximum path delay computed from static timing analysis. For each perturbation, we randomly select a CAB and a target column and move the CAB to either an empty slot or swap with a random CAB in the target column and evaluate the cost using (2). In addition, the evaluation of $\mathrm{hgsw}$is done incrementally so that only the change in the cost function is computed.

VI. Experimental Results

We implemented our algorithms using C++/STL and tested on our analog benchmark circuits shown in Table I. We map these circuits onto three different FPAA chips shown in Table II. Our current FPAA architecture provides more local wires then global, so our physical synthesis process focuses on reducing inter-CAB wires.

Table III presents the % utilization for CABs, i-nets, and x-nets. The maximum longest path delay (xlp) is also presented in this table. For the calculation of CAB utilization, the unused CABs are not considered. The ratio of the number of used components to all available components in a CAB gives the CAB utilization for that CAB. Likewise, the ratio of i-nets (x-nets) used over all available i-nets (x-nets) gives the i-net (x-net) utilization for the CAB in consideration. The average of CAB, i-net and x-net utilization over all used CABs give the overall CAB, i-net and x-net % utilization.

First, the CAB utilization varies little among four different clustering algorithms. This leads us to believe that the CAB utilization is independent of the circuit component ordering and the CAB selection criteria. The CAB utilization improves as the size of the circuit and its corresponding FPAA device increases. Second, we observe that the net-driven method achieves the highest i-net utilization and lowest x-net utilization. Note that 100% i-net utilization is not always possible depending on the circuit structure and CAB component availability. Third, path-driven clustering achieves the best results in reducing the maximum path length. In particular, the path-driven scheme reduces the maximum path length of the second biggest circuit (c19) significantly compared to the other algorithm. We note that the net-driven approach also achieves timing results that are comparable to path-driven, leading us to believe that the minimization of x-net also has an indirect impact on path delay minimization. Lastly, both net-driven and path-driven methods have their own strengths: net-driven method yields better utilization of resources while path-driven method results in better timing results. In addition, the net/path-driven method achieves good tradeoff between resource utilization and timing results.

Table IV studies the impact of various cost factors used in our SA-based refinement. The first three algorithms take only hgsw, mse, or xlp objective into account, whereas the forth algorithm uses a combination of all three objectives and is selected as baseline. First, we obtain 3% more hgswsaving with $\mathrm{hgsw}-\mathrm{only}$algorithm compared to the baseline. However, this saving comes with 438% and 39% increase on mse and xlp cost. Second, the mse saving with $\mathrm{mse}-\mathrm{only}$ algorithm compared to the baseline is almost negligible while the hgsw and xlp cost increase by 16% and 28%, respectively. Finally, the xlp saving with $\mathrm{xlp}-\mathrm{only}$ algorithm compared to the baseline is 15% while the hgsw and mse cost increase by 13% and 64%, respectively. These results reveal that there may be a little improvement for a certain metric if SA focuses only on that metric. However, these individual savings come with huge degradation on other metrics that are ignored. Thus, the combined cost function proves to be the best approach.

VII. Conclusion

This paper focused on making our large-scale floating-gate-based FPAA technology more accessible by providing the first physical synthesis tool. Our analog CAD tool automates the placement of analog circuit components on a target large-scale FPAA. Our placement algorithm incorporates a performance metric that takes into account signal degradation and circuit parasitics under various device-related constraints. Our experimental results demonstrated the effectiveness of our new approaches for solving this new problem.

References

[1] T. Hall, C. Twigg, P. Hasler, and D. Anderson, "Developing large-scale field-programmable analog arrays," in *Proc. Parallel Distrib. Process. Symp.*, 2004, pp. 26–30.

[2] P. Hasler, C. Diorio, B. A. Minch, and C. A. Mead, "Single transistor learning synapses," in *Advances in Neural Information Processing Systems 7*. Cambridge, MA: MIT Press, 1995, pp. 817–824.

[3] J. D. Gray, C. M. Twigg, D. N. Abramson, and P. Hasler, "Characteristics and programming of floating-gate pFET switches in an FPAA crossbar network," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2005, pp. 23–26.

[4] H. Wang and S. Vrudhula, "Behavioral synthesis of field programmable analog array circuits," *ACM Trans. Design Autom. Electron. Syst.*, pp. 563–604, 2002.

[5] S. Ganesan and R. Vemuri, "Behavioral partitioning in the synthesis of mixed analog-digital systems," in *Proc. ACM Design Autom. Conf.*, 2001, pp. 133–138.

[6] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1–12, 1994.

[7] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 1997, pp. 213–222.

[8] M. Pedram, B. Nobandegani, and B. Preas, "Design and analysis of segmented routing channels for row-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1266–1274, 1994.

[9] B. Cochrun and A. Grabel, "A method for the determination of the transfer function of the electronic circuits," *IEEE Trans. Circuit Theory*, vol. CT-20, no. 1, pp. 16–20, Jan. 1973.

[10] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. COM-28, no. 1, pp. 84–95, Jan. 1980.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *Proc. ACM Design Autom. Conf.*, 1997, pp. 526–529.

Virtual Memory Window for Application-Specific Reconfigurable Coprocessors

Miljan Vuletić, Laura Pozzi, and Paolo Ienne

Abstract— **The complexity of hardware/software (HW/SW) interfacing and the lack of portability across different platforms, restrain the widespread use of reconfigurable accelerators and limit the designer productivity. Furthermore, communication between SW and HW parts of codesigned applications are typically exposed to SW programmers and HW designers. In this work, we introduce a virtualization layer that allows reconfigurable application-specific coprocessors to access the user-space virtual memory and share the memory address space with user applications. The layer, consisting of an operating system (OS) extension and a HW component, shifts the burden of moving data between processor and coprocessor from the programmer to the OS, lowers the complexity of interfacing, and hides physical details of the system. Not only does the virtualization layer enhance programming abstraction and portability, but it also performs runtime optimizations: by predicting future memory accesses and speculatively prefetching data, the virtualization layer improves the coprocessor execution—applications achieve better performance without any user intervention. We use two different reconfigurable system-on-chip (SoC) running Linux and codesigned applications to prove the viability of our concept. The applications run faster than their SW versions, and the overhead due to the virtualisation is limited. Dynamic prefetching in the virtualisation layer further reduces the abstraction overhead.**

Index Terms—**Codesign, coprocessors, dynamic prefetching, operating system (OS), reconfigurable computing.**

I. Introduction

Blending two computational paradigms (temporal computation on standard processors and spatial computation in reconfigurable hardware) supported by reconfigurable system-on-chip (SoC) devices [1], [2] is a well-known way to increase performance: critical code sections or entire software functions are mapped to reconfigurable hardware accelerators. When it comes to interfacing the application-specific coprocessors with the rest of the reconfigurable SoC: 1) programmers must be aware of data partitioning and memory transfers and 2) hardware designers have to account for different architectural details of the host platform. The memory transfers can particularly burden the programmer, if shared memory accessible by processor and field-programmable gate array (FPGA) is smaller than a dataset to process.

We introduce an abstraction layer for virtualization of hardware/software (HW/SW) interfacing. A lightweight platform-specific hardware and an operating system (OS) extension reduce the burden of SW programmers and HW designers: 1) programmers can write software that invokes reconfigurable coprocessors as if they were software functions—there is no need for explicit data transfers, passing memory pointers is just enough and 2) designers can write coprocessors that access the user virtual memory through a virtual memory window—there are neither physical constraints on addressing nor on the interface memory size. Our contribution shifts the burden of moving data between processor and coprocessor from the programmer to the OS. Codesigned applications become fully platform independent with only a limited penalty.

Manuscript received April 23, 2004; revised January 23, 2006.

M. Vuleti and P. Ienne are with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne CH-1015, Switzerland (e-mail: miljan.vuletic@epfl.ch; paolo.ienne@epfl.ch).

L. Pozzi was with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne CH-1015, Switzerland. She is now with the Faculty of Informatics, University of Lugano, Lugano CH-6900, Switzerland (e-mail: laura.pozzi@unisi.ch).

Digital Object Identifier 10.1109/TVLSI.2006.878481