Non-Contiguous Linear Placement for Reconfigurable Fabrics

Cristinel Ababei Kia Bazargan
Electrical and Computer Engineering Department
University of Minnesota, Minneapolis, MN 55455
{ababei, kia}@ece.umn.edu

Abstract

We present efficient solutions for the non-contiguous linear placement of data paths for reconfigurable fabrics. A strip-based architecture is assumed for the reconfigurable fabric. A pre-order tree-expression or a general graph is placed in a strip, which can have active and/or inactive pre-placed cores representing blockages and/or cores available for reuse. Two very efficient algorithms are proposed to solve the simpler problem of non-contiguous placement with blockages but without core reuse for tree graphs. The linear ordering obtained with any of the above algorithms is used as input for a third efficient algorithm to solve the problem of non-contiguous placement with both active and inactive cores. A fourth algorithm is proposed to solve the problem of non-contiguous placement with both core and connectivity reuse. Simulations results are reported.

1. Introduction

The manufacturability costs of classic ASICs continue to increase due to the shrinking of transistor size and the increase of circuit complexity. Therefore, the importance of reconfigurable fabrics (e.g., including an FPGA block within an ASIC design) has increased considerably in recent years. A recent example is the hybrid ASIC/FPGA chip by IBM and Xilinx [20]. This solution offers a combination of the performance achievable with ASICs and the flexibility of reconfigurability.

One of the reasons for increased interest in reconfigurable fabrics is the higher flexibility offered by reconfigurability, which allows the implementation of different applications on the same reconfigurable fabric and the upgrade of currently deployed application specific reconfigurable circuits, and significantly decreases the time to market. Another reason for increased attention to reconfigurable fabrics is the continuous increase of the size and performance of reconfigurable fabric, which allows the implementation of more complex and faster applications.

Reconfigurable computing (RC) is used as an alternative to the software implementation of digital media, cryptographic, and compute-intensive algorithms or parts of these (hardware software co-design) due to the superior speed of hardware vs. software. The greater speed comes from the high parallelism and custom datapath widths, which are realizable with RC. An example of an RC platform is the Xilinx Virtex II Pro reconfigurable fabric, which has immersed up to four PowerPC processors [21]. Reconfigurable computing systems (RCS) offer efficient solutions to a variety of problems. For example, they facilitate the implementation as well as the upgrade (possibly via Internet) on the same platform of different features (DSP, cryptography, etc.) for audio and video data streaming portable devices such as cell phones [26], [27], [28]. Embedded systems for automotive applications (e.g., in-car navigating and collision detection systems) is yet another example for use of RCS [29]. Ambient intelligence [22] (a developing metaphor, which describes electronic environments that are sensitive and responsive to the presence of people) is a recent topic supported (especially during development and prototyping [23], [24], [25]) by upgradeable RCS and distributed computing.

A typical RCS architecture may look as that shown in Fig. 1.a. It is similar to the architecture discussed in [1], [2]. Such an RCS has to be supported by physical design CAD tools that shall guarantee fast run-times. Such tools are indispensable for the generation of (partial) reconfiguration bits in very short times. To achieve short run-times one has to innovate at both architectural and algorithmic levels. At the architectural level, we propose that the reconfigurable processor unit (RPU) be divided into *physical strips*. A physical strip represents a horizontal strip of the FPGA chip. Each strip could be paired at both endings with I/O memories, which serve as FIFO buffers for data to be processed or for results. The height of a strip (and therefore the total number of strips on the chip; see Fig. 1.b) can vary depending on the data width to be processed. The motivation behind such RPU architecture is that it simplifies the physical design. The placement of cores[1] (i.e., "hardware cores" stored in libraries, such as multipliers and adders) is now linear as opposed to typical 2D approaches [4], [2]. This simplification[2] facilitates faster placement and

[1] Also referred to as reconfigurable functional unit operations (RFUOPs) in frameworks like those in [4], [2].

[2] It is worth mentioning that such architecture would not impair the achievable performance but would only possibly require larger areas, as experiments reported in [10] showed.

routing engines and easier integration of techniques like the virtual sockets described in [5] (physical design becomes more platform-based) or the core communication interface in [13].

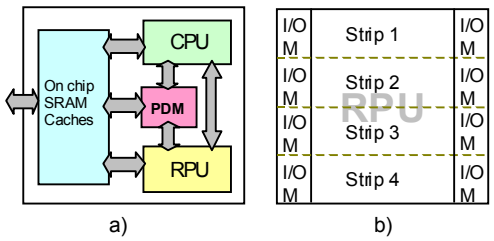

Fig. 1 a) The RCS architecture b) The RPU divided into four strips

1.1 Previous Work and Current Challenges

In partial dynamic reconfiguration only the area required by the new configuration is reconfigured based on events[3]. The rest of the RPU remains intact. In this way reconfiguration bitstreams are smaller and hence the reconfiguration time is smaller. Static and full dynamic configuration modes are supported by mature CAD tools, which cover all steps in the design cycle (from high-level design specifications down to placement and routing and then static configuration of a single or multiple contexts). However, the partial run-time reconfigurable model lacks the design tools for partial reconfiguration, which makes practical implementations a challenging task, even though the partially reconfigurable devices (e.g., Xilinx Virtex [18]) are already available. We note the practical lack of a coherent design flow and an RCS platform that shall provide real-time reconfiguration bits generation and partial dynamic reconfigurability. However, many RCS architectures have been proposed with notable performance limited to specific applications [8] [9].

1.2 Our Work

We focus on the physical design CAD tools aspect. In particular, we concentrate on two of the three placement problems, which we envision to appear in a partial dynamically reconfigurable computing system such as that shown in Fig. 1. The first placement problem in such a framework is when a new expression is implemented into an empty strip. In this case the placement is *static linear placement* (because the placement can be done during compilation) similar to [10]. When a new expression is to be implemented into a strip, which already hosts a previous expression or the latter needs to be augmented with additional functionality the placement problem becomes a *non-contiguous linear placement*. In this case only certain areas within the strip are available for the placement of new cores and therefore the classical linear placement has now additional constraints under the form of blockages (i.e., obstructions). The third kind of placement represents the case when previously placed cores (possibly interconnected) are still spread into a strip and decisions have to be made about which ones to be used for the expression to be placed (therefore to reduce reconfiguration time), which ones to be kept for future expressions, or which ones to be erased or relocated inside the same strip. We call this *dynamic* (or real-time) *linear placement* because decisions, which directly impact the placement performance, have to be made in some cases at run-time.

We propose solutions for the second and third kinds of placement. Our algorithms are intended for fast and close to optimum implementation of relatively simple kernels[4] (which can be the result of a dynamic hardware/software partitioning encountered in HW/SW co-design [3]). In a typical RCS, the application is partitioned into loops (a loop has more kernels) implemented into hardware and software. When a kernel is implemented in hardware, its actual implementation (placement, core shapes, etc.) is done such that the execution time for the whole application is minimized. Our algorithms can be used either for fast solution-space search during compile-time (to achieve best configurations/contexts to be swapped during execution) or for real-time placement in cases when the hardware-software partitioning is done in real-time during execution (in this case our algorithms would be behind the physical design manager – PDM – which can be another dedicated processor, in Fig. 1.a). We exploit re-usability in a partially reconfigurable framework, which is a characteristic of the emerging platform-based design methodologies. This is in contrast to, for example, [10] where the focus is on rather operation-merging-based regular data-path synthesis and optimization, which is suitable for customized implementation of data-paths in a static framework.

1.3 Problem Formulation

We now state the assumptions that we make and present the formulation of the non-contiguous placement problem. The framework diagram of the non-contiguous placement is presented in Fig. 2.

In order to simplify the problem we make the following assumptions:

- The RPU is an island-like FPGA chip containing 64x64 CLBs excluding the I/O memories on the left and right sides of the chip (see Fig. 1.b). Strips are horizontally considered and the height of each strip is determined by the data width to be processed. The height should accommodate the tallest core available in the library. The "Library" box in

[3] One can also talk about static configuration - the whole RPU area has to be reconfigured during each reconfiguration or full dynamic reconfiguration - more configurations are stored in the FPGA or in a cache on-chip memory and they are switched during execution as response to requests [6]. In this work our focus is however on partial dynamic reconfiguration only.

[4] In this work, a kernel is a pre-order binary expression (i.e., a tree graph) or a general directed graph.

Fig. 2 represents the list of cores and their characterization obtained with JBits [7].

❑ Wire delays are estimated considering that every connection was routed considering best possible combination of wire segments. A look-up table is created, which has delay entries for every possible distance. Note, that in this way the delay estimation is optimistic. The "Delay Architecture" box in Fig. 2 represents the delay look-up table.

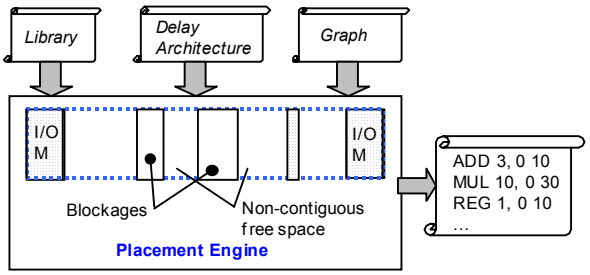

Fig. 2 Non-contiguous linear placement diagram

❑ Coordinates of pre-placed cores are given. *Active* pre-placed cores are considered as blockage areas/intervals. *Inactive* pre-placed cores are available for reuse. The "Graph" box in Fig. 2 represents the pre-order binary tree or general graph, which has to be placed together with the information about active (i.e., blockages) and inactive cores.

❑ Primary inputs (PIs) and primary outputs (POs) are placed at both sides (left and right) of the strip for congestion minimization.

The formulation of the non-contiguous placement with blockages problem is as follows:

***GIVEN**: The delay look-up table (architecture), the library of cores together with their characterization, the tree or general graph (as data flow graph - DFG), and the location of pre-placed cores inside the strip.*

***OBJECTIVE**: Overlap-free linear placement (x coordinates) of all cores of the DFG such that the Wire-Length (WL), the delay at the outputs, and the max-cut at any column (congestion) are minimized.*

2 Non-contiguous Linear Placement for Tree Graphs

We present three different algorithms for solving the non-contiguous linear placement for tree graphs.

2.1 Algorithm H1

The first proposed placement algorithm is based on the idea of a heuristic partitioning algorithm presented in [11]. We adopted and applied it to our linear non-contiguous placement problem because it fits well the purposes of max-cut and WL minimization. In what follows we describe how our algorithm works. The pseudo-code of the first algorithm is shown in Fig. 3.

First, we build the *EV-matrix*, which is an $m \times n$ matrix where m – the number of rows – is the number of edges in the directed tree-graph and n – the number of columns – is the number of nodes. An element $a(i, j)=1$ in the matrix is non-zero if the *j-th* node is a terminal of the *i-th* net. If a node is not a terminal for a net, the corresponding EV-matrix element is zero. For example, Fig. 4 shows the binary tree[5] for the expression Tg01.exp and the EV-matrix built based on the counting of all nodes and nets.

```
Input:
        G(V,E)
        Library of cores and delay info
Step:
1.      Build EV-matrix
2.      Transform EV-matrix into band-matrix
3.      Greedy overlap-free placement
```

Fig. 3 Algorithm H1

Then, the EV-matrix is transformed into an as-close-as-possible band-form matrix using a procedure similar to the one in [11]. We denote this as *B(EV)-min* problem (i.e., minimization of the bandwidth of the EV-matrix problem). The procedure uses row and column flips only and is based on a sorting algorithm. The goal of getting the matrix to a band-form (which translates into an optimal linear ordering) serves two objectives:

1. Cutsize minimization – by having all 1's in the matrix clustered along the diagonal shown with dashed-line in Fig. 4.b, the cutsize (the number of nets cut by a vertical cut applied between any two consecutive nodes in the linear arrangement) is minimized everywhere in the linear arrangement.

2. WL minimization – by minimizing the bandwidth (maximum x distance spanned by any of the nets) of the EV-matrix, the total wire-length of all nets is minimized.

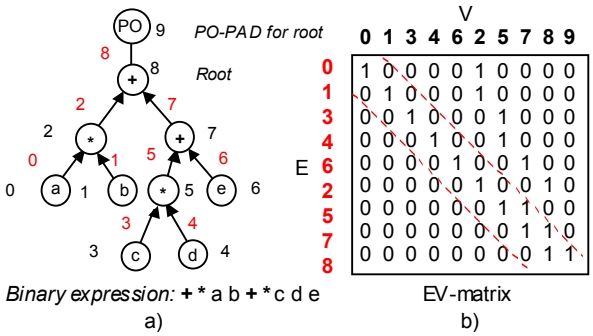

Fig. 4 a) Binary expression tree b) The EV-matrix with nodes placed on columns such that the 1's for PI nodes are in the top-left corner and the 1 for PO-pad node is in the right-bottom corner

We would like to note that our EV-matrix is different from the VV-matrix, which is commonly used to represent

[5] Note that an extra node - PO pad - was inserted. This is done for every tree with the purpose of connects the root to the I/O memories and for wirelength computations.

the incidence-matrix of a graph, and for which the BANDWIDTH problem is known to be NP-complete in general as well as for trees with maximum degree of three or more [12].

The congestion is indirectly minimized by minimization of the cutsize as well as the assignment of the PI/PO pads evenly to the left and right ends of the strip.

After bandwidth minimization, we greedily place the cores from right-to-left in the strip, based on the final ordering of nodes in the EV-matrix, such that no overlaps exist and blockages are skipped. That is performed under the assumption that the PO-pad is assigned to the right end of the strip; otherwise the placement is performed from left-to-right. The run-time complexity of algorithm H1 is *O(a lg a), a=max(m,n)* and it is dominated by the quicksort algorithm behind the procedure in Step 2.

2.2 Algorithm H2

The idea of the second heuristic is also based on the minimization of the bandwidth of the EV-matrix. However, this time we do not need to actually build the EV-matrix but we rather work directly on the binary-tree graph in a top-down approach. The main steps of our algorithm are shown in Fig. 5.

Input:	
	G(V,E)
	Library of cores and delay info
Step:	
1.	Post-order traversal to compute quantities of interest
2.	Rank assignment to nodes by linear ordering
3.	Greedy overlap-free placement

Fig. 5 Algorithm H2

First, a post-order walk is performed in order to compute the following interval variables for every node:

1. Latest arrival time, *d* (delay), up to the current node starting at any of the leaf nodes in the corresponding sub-tree. Initially, all delays are computed considering zero-delay for all wires. The delay *d* will be actually an interval (d_{min}, d_{max}) because every core in the library can have multiple shapes (widths and heights) with different delays.

2. Longest (maximum) path-width, *pw*, up to the current node starting at any of the leaf nodes in the corresponding sub-tree. A path-width is the sum of all widths of all cores along a path from a leaf-node up to the root-node of a sub-tree. The maximum path-width will also be computed as an interval (pw_{min}, pw_{max}) because the width of every core is at its turn an interval of discrete values. This variable is useful because stores the total width of the path, if all modules along the path are placed contiguously.

3. Volume-width of a sub-tree, *v*, represents the sum of all widths of all cores in sub-tree. The volume will also be computed as an interval (v_{min}, v_{max}). Volume-width gives the total width of the sub-tree, if all modules in the sub-tree are placed contiguously.

For example, Fig. 6 illustrates a generic node of a tree with the critical path having the latest delay *d*, the maximum path-width *pw*, and the volume-width *v* shown for the right sub-tree.

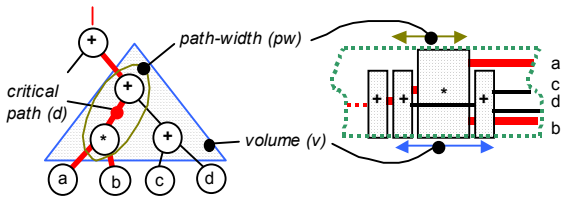

Fig. 6 Illustration of path delay, path-width, and volume

Then a linear ordering (i.e., numbering, counting) of all nodes is performed by assignment of ranks starting with the root node (and its associated PO-pad) and recursively continuing to the left (decreasing of the rank) with the left sub-tree and to the right (increasing of the rank) with the right sub-tree. This step is illustrated on a very simple example in Fig. 7 (corresponding to the example in Fig. 4, with the assumption that all cores in the library have unique shapes).

The rank of the root is equal to the number of nodes in the left sub-tree of the root. The rank of the PO-pad is the rank of the root plus one (i.e., PO-pad is next to the right of the root). Ranking of nodes in either of the left or right sub-trees is done in a "smaller-volume first" order of their own sub-trees. For example, in Fig. 7, node 6 is ranked such that it is to the left of node 5. That is because the sub-tree {6} undergoes first the process of ranking because this sub-tree has a smaller volume than the sub-tree containing node 5, which is {5,3,4}. If both sub-trees of the generic node have the same volume, the ranking is decided based on max path-width or delay. At the end of this step we have obtained a linear numbering of nodes as that shown in the shaded area in Fig. 7. As pointed out in the previous section, this linear ordering is very important because it is a direct measure of the quality of the final placement in terms of total wire-length and maximum cutsize.

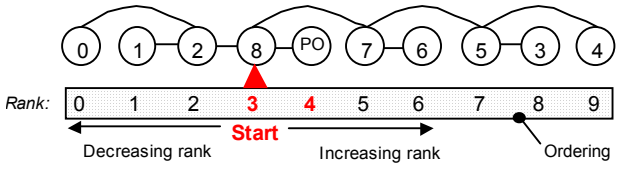

Fig. 7 Illustration of rank assignment in order to obtain the linear numbering for the example in Fig. 4

Finally, a post processing is performed in order to assign *x* coordinates to all cores from right-to-left such that there are no overlaps and all blockages are skipped. During this step PIs (i.e., leaf nodes) and the PO-pad will be assigned to the left or right of the strip depending on which one is closer, in order to minimize the wire-length. When cores have multiple shapes, then decision about which shape for a given core to be used has to be made. The shape assignment is done dynamically such that the delay of the most critical

path is minimized while the total area of the cores off critical path is minimized. The run-time of the algorithm H2 is *O(n)*.

2.3 Algorithm H3

In this section we address the third kind of non-contiguous placement. In this case, pre-placed cores may be inactive and available to be used as resources for the expression to be placed. The goal here is to re-use as many inactive cores as possible, hence to reduce the reconfiguration time, while minimizing the same objective functions: wire-length, the delay at the output of the expression tree, and the max-cut at any *x*. In this case the problem is more difficult because the way the inactive cores are placed on the strip can lead to increase in wire-length and delay as well as of max-cut. The number of inactive cores usually is not equal to the number of cores of the expression to be placed. Therefore, we have to decide on which inactive cores to re-use and which cores to be placed to match with. Other important factors are the "distribution" of inactive cores on the strip and their type (e.g., multipliers, adders, etc.). If all inactive cores are "flushed" to an end of the strip, then their re-use is likely to be more difficult compared to the case when they are uniformly distributed on the strip between blockages. If, for instance, all inactive cores are of one type (e.g., multipliers) and all cores to be placed are of another type (e.g., adders), then all inactive cores have to be treated as blockages or to be "overwritten".

The main steps of algorithm H3 are shown in Fig. 8. First, a linear ordering is found for the expression to be placed using one of the algorithms presented in previous two sections. Because the linear ordering directly affects wire-length and maximum cutsize, we would first like to find a very good linear ordering irrespective of the type and distribution of the inactive cores[6].

Then, perform a maximum matching between the inactive cores and the cores to be placed. Maximum means that we look for re-using as many inactive cores as possible in a manner which will also lead to a minimum perturbation of the linear ordering obtained during Step 1, hence a minimum deterioration of the wire-length, delay, and max-cut. This step is implemented as follows. We model the inactive cores and the empty intervals between them as nodes of a linear graph denoted *strip_g*. The cores to be placed in the linear ordering obtained during the first step represent a second linear graph denoted *linord_g*. We then perform a *linear mapping* between the two graphs.

[6] Delay is also affected by the linear ordering. The linear ordering has to ensure a "straight" (from left to right or right to left) rather than a "back-and-forth" critical path in order to achieve delay minimization. This is true irrespective of the fixed locations of the PIs and POs. However, blockages can have a certain impact (depending on the delay-distance relationship – captured in the look-up table in our case) on the overall delay as well when the final placement will be obtained by flushing all cells to the right or to the left.

Input:
- G(V,E)
- Library of cores and delay info
- Locations and types of pre-placed cores

Step:
1. Use algorithm H1/2 to find linear ordering
2. Max-matching between pre-placed and to be placed cores
3. Greedy placement of left-over cores

Fig. 8 Algorithm H3

The mapping is performed by first partitioning (i.e., dividing) the larger graph into a set of blocks equal to the number of nodes in the smaller graph. Then we match every node in the smaller graph with its corresponding block in the larger graph. For example, Fig. 9 shows the example from Fig. 7 to illustrate how the linear mapping is performed. For example, the node 8 in *linord_g*, which is an adder is mapped to the nodes 3 and 4 in *strip_g*, which are an inactive adder core placed at x=17 and an empty interval [19,20]. This means that the adder 8 from *linord_g* will be most likely matched either with the inactive adder 3 or with the empty interval 4 in *strip_g*. A core to be placed, as a node in *linord_g*, can also be matched with nodes, which are first order[7] (or higher, depending by how much the linear ordering is allowed to change) neighbors of the nodes to which the core is mapped by the linear mapping. For example, the multiplier 5 of *linord_g* can be matched with the inactive core 6 of *strip_g* (see Fig. 9).

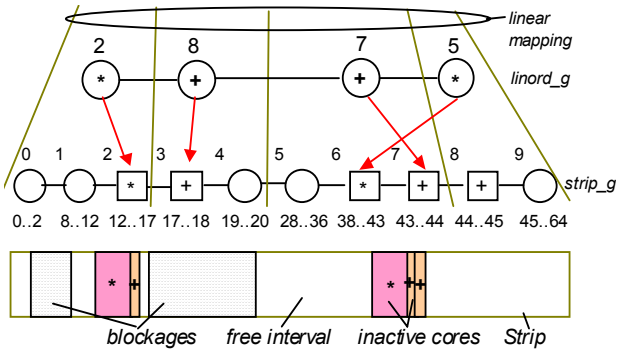

Fig. 9 Illustration of the linear mapping between the two linear graphs for the example in Fig. 4

After mapping and matching, some of the cores of the currently placed expression (nodes of *linord_g*) may not be matched. Therefore, as a last step we greedily match those cores to empty intervals as close as possible to those, which are already matched, in a manner that preserves the initial linear ordering.

Note that our method is a constructive rather than an iterative one. Hence, short run-times are facilitated. The run-time complexity of algorithm H3 depends on which algorithm is used in Step 1.

[7] This is a control parameter that allows the user to tune the algorithm.

3 Non-contiguous Linear Placement for General Graphs

We now present a solution for the non-contiguous linear placement for general graphs, which can have multiple primary outputs. Our goal in this case is similar to that of the problem statement in Section 2.3. We would like to reuse as many inactive cores as possible in order to decrease the amount of reconfiguration bits and hence the reconfiguration time. However, we further consider the interconnection reuse as well. In other words, fully pre-placed general graphs are considered as inactive in a strip. We would like to reuse inactive cores together with their connectivity. That would require less routing effort (and possibly even smaller amounts of reconfiguration bits) for the new graph to be placed. Fig. 10 describes what the optimal core and interconnect reuse is for a very simple example.

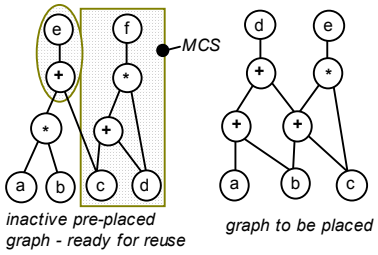

Fig. 10 Illustration of the maximum common sub-graph (MCS) match for both core and connectivity reuse

The pseudo-code of our algorithm is shown in Fig. 11.

Input:	
	G1(V1,E1), G2(V2,E2)
	Library of cores and delay info
	Locations and types of pre-placed cores
Step:	
1.	Use algorithm H1 to find linear ordering
2.	MCS between pre-placed and to be placed cores with connectivity considered
3.	Greedy placement of left-over cores

Fig. 11 Algorithm H4

First, a linear ordering (similarly to the first step of the algorithm in Section 2.3) of the graph to be placed is found using the same procedure as the one described in Section 2.1, which applies for general graphs as well.

Then we find the maximum common sub-graph (MCS) between the pre-placed inactive graph and the graph to be placed. For this purpose we employ a specialized algorithm, developed by Foggia et al as an extension of the algorithm presented in [14] and available at [19]. In its simpler version the MSC algorithms finds the maximum connected sub-graph and in its more complex form (requires longer run-times) the MSC algorithm finds a maximum disconnected sub-graph (e.g., a forest of trees) like for instance the example shown in Fig. 10. The MCS algorithm finds the maximum number of cores (similarly to [16]), which can be reused as well as the connectivity between these cores (similarly to [17]).

In the last step, the left-over cores of the graph to be placed are greedily placed into empty intervals as close as possible to cores, which are already placed (coordinates determined by the MCS), in a manner that preserves the linear ordering obtained in the first step. The run-time complexity entirely depends on the MCS algorithm, which is $O(b\ (n_1 + n_2))$, *b=number of branches* involving any two nodes of the two matched graphs.

4 Simulation Experiments

We now report simulation results obtained with our algorithms. The first three heuristic algorithms (described in Section 2) are used to test a set of six randomly constructed binary expressions[8]. The simulations results are shown in Table 1.

We also implemented a Simulated Annealing (SA) algorithm for delay minimization[9]. We see that, the timing-driven Simulated Annealing engine cannot improve on the delay results obtained with H1 and H2. The reason for that is that the critical path usually starts at the left and ends at the right. This makes, in the case of trees, for the ordering of cores (i.e., their counting) to be more important than the core coordinates.

Comparing the first two heuristics, it can be seen that the delay obtained with H2 is similar to the delay obtained with the H1 but the wire-length is improved. That is because of the better bandwidth minimization obtained with H2. The last three columns of Table 1 present the simulation results obtained with H3. When inactive cores are reused, more factors come into play and contribute to the tradeoffs existing between wire-length, delay, and the number of available inactive cores, which are actually used. We observed in our experiments a stable tradeoff between how well the best linear ordering (obtained in the first step of H3) is preserved and the number of inactive cores, which are actually reused. The more inactive cores are reused (which eventually will translate in faster reconfiguration times) the worse the preservation of the linear ordering is (which translates in slightly higher delay and wire-length).

Table 2 presents simulation results obtained with the algorithm presented in Section 3. We tested our algorithm on a set of selected basic blocks (as data flow graphs) of the Honeywell and MediaBench benchmarks [15]. For comparison purposes, we first place each graph contiguously (each graph is placed without reusing any cores). Then, every graph - starting with the second - is placed considering as pre-placed reusable graph the previous one. This is performed with the MCS algorithm set

[8] All tree and general graphs together with the C++ implementation of all algorithms presented in this work are available for download at [30].

[9] There is no prior work on non-contiguous linear placement to which we can compare our results. The only work somewhat similar to our work is [10]. However, the placement algorithm in [10] solves the problem of static linear placement, which is different from the non-contiguous linear placement problem tackled in this work. Additionally, wire-length is not considered in [10].

to search for the maximum common disconnected sub-graph (maximum core and connectivity reuse but longer run-times) or set to search for the maximum common connected sub-graph (lower core and connectivity reuse but much faster). We can see that in the first setting the percentage of cores and connectivity that are reused is bigger (up to 74% core and up to 36% connectivity reuse) than in the second setting, but at the expense of longer run-times. Bigger core and connectivity reuse translates into smaller amounts of reconfiguration bits, hence shorter reconfiguration times. However, the longer run-times (entirely due to the MCS algorithm) suggest the use of our algorithm - with the first setting - for optimization placement at compile time.

5 Conclusion

We presented efficient solutions for the non-contiguous linear placement problem for reconfigurable computing. Our placement algorithms are based on a heuristic for B(EV)-min (minimization of the bandwidth of the EV-matrix of the expression graph), which translates into direct minimization of the cutsize (congestion) and wire-length and indirect minimization of timing. When inactive cores are reused our algorithms look for finding the maximum matching between the inactive cores and those, which have to be placed. Core and connectivity reuse can be as much as 74% and 36% respectively.

Current and future work focuses on developing a direct delay minimization integrated into the presented algorithms. That should be in the form of either net-based (slack) or path-based (k-most critical paths). A better delay estimation possibly by means of integrating placement with routing is one more way of improving on the placement methodologies for reconfigurable computing.

Acknowledgment

Pasquale Foggia of the Federico II University of Naples, provided timely clarifications about the MCS algorithm used in Section 3.

Table 1 Simulation results for tree graphs

		SA			H1			H2			H3		
Circuit	Nodes	Delay	WL	CPU (s)	Delay	WL	CPU (s)	Delay	WL	CPU (s)	Delay	WL	CPU (s)
Tg01	10	1.16	169	16	1.16	191	0	1.16	187	0	1.21	152	0.015
Tg02	20	1.8	336	60	1.8	328	0.015	1.8	330	0	1.87	350	0.015
Tg03	12	1.58	227	16	1.58	235	0.015	1.58	219	0.015	1.58	224	0.015
Tg04	20	2.6	369	68	2.6	371	0.015	2.6	289	0	2.64	304	0.015
Tg05	20	2.6	302	73	2.6	314	0.015	2.59	293	0.015	2.6	309	0.015
Tg06	28	2.66	371	44	2.66	386	0.015	2.65	367	0	2.66	416	0

Table 2 Simulation results for general graphs

			Contiguous Placement			Non-Contiguous Placement (disconnected MCS)				Non-Contiguous Placement (connected MCS)			
Circuit placed (reused)	Nodes / Nets	PI / PO	Delay	WL	CPU (s)	Delay	WL	CPU (s)	Reuse Core / Connect	Delay	WL	CPU (s)	Reuse Core / Connect
Honeywell-intfc01 (Honeywell-intfc01)	16/13	4/3	1.079	255	0	1.079	255	0	100 / 100	1.079	261	0.016	62.5 / 75
Dft (Honeywell-intfc01)	19/12	4/7	0.955	386	0	0.979	382	0.094	52.63 / 33.33	0.985	412	0.015	15.79 / 13.33
Honeywell-versatil (Dft)	27/20	6/7	1.038	430	0.015	1.078	433	0.469	51.81 / 28	1.075	425	0.015	22.22 / 20
Honeywell-intfc02 (Honeywell-versatil)	27/21	6/6	1.467	458	0	1.508	481	18.6	74.07 / 36	1.467	460	0.032	18.51 / 16
Honeywell-fft01 (Honeywell-intfc02)	31/23	8/8	1.411	595	0.015	1.414	564	187.3	51.61 / 24.13	1.411	596	0.016	16.12 / 13.79
Honeywell-fft02 (Honeywell-fft01)	31/24	5/7	1.057	471	0	1.072	471	414.7	45.16 / 23.33	1.057	479	0.032	16.12 / 13.33
MediaBench-jpeg (Honeywell-fft02)	35/27	9/8	1	650	0	1.031	607	597.2	54.28 / 27.27	1.009	643	0.015	11.42 / 9.09

References

[1] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *ACM/IEEE Design Automation Conference (DAC)*, 2000, pp. 507-512.

[2] K. Bazargan, S. Ogrenci, and M. Sarrafzadeh, "Integrating Scheduling and Physical Design into Coherent Compilation Cycle for Reconfigurable Computing Architectures", *ACM/IEEE Design Automation Conference (DAC)*, 2001, pp. 635-640.

[3] K. B. Chehida and M. Auguin, "HW/SW Partitioning Approach for Reconfigurable Systems Design", *International Conference on*

Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2002, pp. 247-251.

[4] G. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware", *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 72-81.

[5] M. Dyer, C. Plessl, and M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex", *International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 292-301.

[6] Z. Li, K. Compton, and S. Hauck, "Configuration Caching Techniques for FPGA", *IEEE International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2000, pp. 22-36.

[7] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java based interface for reconfigurable computing", *Military and Aerospace Programmable Logic Devices International Conference (MAPLD)*, 2002.

[8] A. DeHon and J. Wawrzynek, "Embedded Tutorial: Reconfigurable Computing: What, Why, and Implications for Design Automation", *ACM/IEEE Design Automation Conference (DAC)*, 1999, pp. 610-615.

[9] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, 1998, 86(4): 615-638.

[10] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs", *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 1998., pp. 123-132.

[11] S. -W. Cheng and K. -H. Cheng, "ENISLE: an intuitive heuristic nearly optimal solution for mincut and ratio mincut partitioning", *International Symposium on Circuits and Systems (ISCAS)*, 2001, pp. 167-170.

[12] J. Diaz, J. Petit, and M. Serna, "A Survey of Layout Problems", *ACM Computing Surveys*, Sept. 2002, pp. 313-356.

[13] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans, "Remote and Partial Reconfiguration of FPGAs: Tools and Trends", *Reconfigurable Architectures Workshop (RAW)*, 2003.

[14] P. Foggia, C. Sansone, and M. Vento, "An Improved Algorithm for Matching Large Graphs", *The 3rd IAPR-TC15 Workshop on Graph-based Representations*, Ischia, 2001.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *ACM/IEEE International Symposium on Microarchitecture*, 1997.

[16] S. O. Memik, G. Memik, R. Jafari, and E. Kursun, "Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs", *ACM/IEEE Design Automation Conference (DAC)*, 2003, pp. 604-609.

[17] N. Moreano, G. Araujo, Z. Huang, and S. Malik, "Datapath Merging and Interconnection Sharing for Reconfigurable Architectures", *International Symposium on System Synthesis (ISSS)*, 2002, pp. 38-43.

[18] Xilinx Inc., *Virtex II FPGA Advance Product Specification*, 2001, Available at: www.xilinx.com.

[19] http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html

[20] www-3.ibm.com/chips/products/asics/products/cores/efpga.html

[21] http://www.xilinx.com/products/tables/fpga.htm#v2p

[22] http://www.research.philips.com/InformationCenter/Global/FArticleSummary.asp?lNodeId=931#ambintel

[23] http://research.microsoft.com/easyliving/

[24] http://architecture.mit.edu/house_n/

[25] http://www.awarehome.gatech.edu/

[26] http://www.qstech.com/tech_products.htm

[27] http://www.picochip.com/

[28] http://www.xilinx.com/apps/epld.htm#CoolRunner

[29] http://www.xilinx.com/publications/products/sp2e/wp_pdf/wp153.pdf

[30] http://www.ece.umn.edu/users/ababei