TEMPORAL PARTITIONING AND SEQUENCING OF DATAFLOW GRAPHS ON RECONFIGURABLE SYSTEMS

Christophe Bobda
Heinz Nixdorf Institute/Paderborn University
Fuerstenallee 11, D-33102 Paderborn, Germany
bobda@upb.de

Abstract FPGAs(Field Programmable Gate Arrays) are often used as reconfigurable device. Because the functions to be implemented in FPGAs are often too big to fit in one device, they are divided into several *partitions or configurations* which can fit in the device. According to dependencies given in the function a Schedule is calculated. The partitions are successively downloaded in the device in accordance with the schedule until the complete function is computed. Often the time needed for reconfiguration is too high compared to the computation time [1, 11]. This paper presents a novel method for the reduction of the total reconfiguration time of a function by the generation of a minimal number of configurations. We present the framework that we developed for the fast and easy generation of configurations from a function modeled as DFG (dataflow graph).

Keywords: Reconfigurable Computing, Temporal Partitioning, Fast Synthesis

1. INTRODUCTION

Combining the dataflow aspect and the inherent parallelism which characterizes some classes of functions, FPGAs can be used to implement those functions more efficiently than CPUs and more flexible than ASICs. FPGAs have successfully been used to provide fast computation in many application areas including text and image processing and floating point computation.
Reconfiguration is used to implement functions of any size in FPGAs with small capacity. This is done by dividing the functions in partitions which fits in the FPGAs. The partitions are implemented as configurations or bitstreams which are successively downloaded into FPGAs to compute the desire functions. This process is usually called *temporal partitioning* or *temporal placement*. Most of the works done on temporal partitioning assume the FPGAs to be either partially reconfigurable or time multiplexed. That means many configurations

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35599-3_29 are stored on the chip and can be quickly (in nano seconds) downloaded in the FPGAs when needed. But the reality is different. Most of commercials FPGAs are neither time multiplexed nor really partial reconfigurable. For this purpose, many solutions proposed are not easily applicable. In practice high level tools are used to capture and solve problems which are geometrically based. The consequence is an inefficient use of the FPGA capabilities like the regularity structure. This paper examines the use of non-partial reconfigurable FPGAs to implement large functions. The goal is to minimize the reconfiguration overhead which is always too high compare to the computation time of function in FPGAs. We have developed a framework for the capture, the temporal partitioning and the generation of configurations from a given function in form of a DFG. Our framework is based on the JBits[6] API. We reduce the temporal placement problem to the placement of a small amount of modules in different configurations by dealing with cores. The rest of the paper is organized as follows: Section 2 presents the main work which has been carried in the temporal placement and temporal partitioning. In section 3 the formulation of the temporal placement problem based on the definitions already given in [12, 14] is given. The realization of overlapping between cores is shown in section 4. Section 5 presents the list scheduling based temporal partitioning method. To illustrate the concept of section 5, the partitioning method is applied on a real life problem in section 8. Section 9 concludes the paper and gives some directions for future work.

2. RELATED WORK

The most used and perhaps the simplest approach to solve the temporal partitioning problem is the (enhance)list scheduling [11, 3, 10, 4, 13]. This method first places all the nodes of a DFG representing the problem to be solved in a list. Partitions(configurations) are built stepwise by removing nodes without predecessor in the list and allocate them to a partition until the size of the partition reached a limit(the size of the FPGA). Integer linear programming(ILP) [10, 5, 8] can be applied to solve some optimization constrains like timing. Other methods like network flow [9] and genetic algorithm [15] are also applied to find temporal partitions and optimize parameters like communication cost and configuration latency.

In order to reduce the reconfiguration overhead, Pandey et al[11] suggested the reduction of number of partitions with resource sharing. For a new partition, a minimal resource set is chosen and a partition is built upon this resource set. Trimberger[13] proposed a time multiplexed FPGA architecture in which configurations can be stored directly on the chip. Reconfiguration is done in micro cycle. In a micro cycle a new configuration is downloaded from the on-chip memory. The CLBs(configurable logic block) contain micro-registers to temporally hold results of previous computation step when switching from one configuration to another.

The geometrical view of the partial reconfiguration is considered in some extent in [12, 14]. The space and temporal placement of computational nodes of a DFG representing the problem to be computed in the FPGA is mapped, for example, to a packing problem[12]. Problems with high reconfiguration time(which are the focus of this paper) have been defined as a matter of future work. In practice high level description languages and tools are used to specify and solve a problem with important geometrical aspects. This leads to a non efficient use of the regularity structure of FPGAs[2] as well as waste of resource. We present an approach for the minimization of reconfiguration. For the Xilinx Virtex FPGA family that we use as RPU(reconfigurable processing unit), cores can be generated in the JBits environment. This java-based interface provides some basic functions like adders, comparators, subtracters, multipliers, multiplexers which can directly be placed on different locations on the FPGA and routed together to build complex functions.

3. PROBLEM FORMULATION

This section deals with definition and formulation of the problem to be solved. Since the definitions provided in [12, 14] are the most adapted for the temporal placement, we choose to adopt it and adjust them to fetch our considerations.

Definition 1 (Dataflow Graph) *Given a set of tasks $T = \{T_1, ..., T_k\}$ a dataflow graph is a directed acyclic graph $G = (V, E)$, where $V = T$. An edge $e = (T_i, T_j) \in E$ is defined through the (data)dependence between task T_i and task T_j.*

Each node T in the DFG is equivalent to a core C which can be placed on the RPU H with length h_x and width h_y.

Definition 2 (Temporal placement) *Given a DFG $G = (V, E)$ and a RPU H with the size (h_x, h_y), a temporal placement is a three dimensional vector function $p = (p_x, p_y, p_t) : \Omega \rightarrow N^3$, where the values $p_x(C)$, $p_y(C)$ denote the coordinates of the lower left position of the core C associate to a task $T \in V$ on the RPU. The core C occupied the space $[p_x(C), .., p_x(C) + w_x(C)]$ in the x direction, $[p_y(C), .., p_y(C) + w_y(C)]$ in the y direction and $[p_t(T), .., p_t(T) + w_t(C)]$ in the time dimension.*

Figure 1 illustrate a temporal placement of a problem graph with three cores(+),(*) and (>) representing tasks T_1, T_2 and T_3) with dependencies $(T_1 \rightarrow T_2)$ and $(T_1 \rightarrow T_3)$ on a partial reconfigurable FPGA.

In our formulation of the temporal partitioning, modules are allowed to overlap in space and time. Reconfiguration happens in today's FPGAs by the complete replacement of configuration, a process which is too costly compare to the

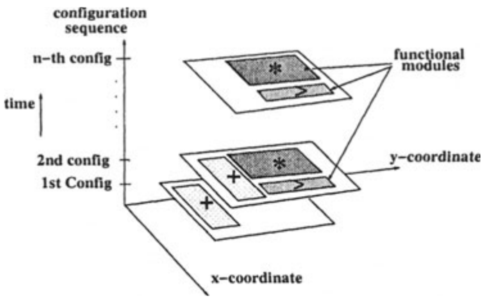
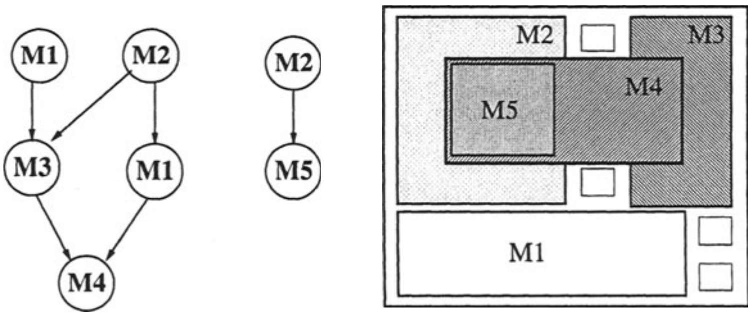


Figure 1. A temporal Placement

computation time[11, 1]. To avoid such overhead, we exploit the free resources of modules already selected to run in the FPGA in a time interval to implement the modules for other computation. This process require no additional overhead, because only free resource are use.

In fig.2, all the modules could not be placed simultaneously on the device if overlapping was not allowed.



(a) A DFG (b) Temporal Placement with overlapping

Figure 2. A DFG and the corresponding temporal placement with ressource sharing

4. IMPLEMENTATION OF OVERLAPPING CORES AND RESSOURCE SHARING

If two cores C_1 and C_2 share the same resource C_0 and if the free resource in core C_0 are enough to implement the functionality of C_0, then C_0 can be shared by C_1 and C_2 without additional resource. For example, a Virtex register implemented in the JBits use only the F_1 and G_1 inputs of the slice 0 or 1 of a

```
ALGORITHM I: generate_partitions(DFG G, DEVICE dev)
1 BEGIN;
2 list_nodes := generate_nodes_list_with_priority(G);
3 WHILE (!list_node.empty()) DO
4   allocated_list.reset();
5   FOR(i := 1; i < list_nodes.size();i++)
6     chosen_node := node_list[i];
7     best_node := get_best_overlapping_node();
8     IF (best_node = NULL)
9       IF (allocated_list.size + chosen_node.size < dev.size)
10        allocated_list.insert(chosen_node);
11        list_node.remove(chosen_node);
12      ELSE
13      implement_overlap(chosen_node, best_node);
14 list_partition.insert(allocated_list);
15 END;
```

Figure 3. The temporal partitioning algorithm

CLB(Configurable logic Block). Its outputs are the X_q and Y_q outputs of the corresponding slice. The resource $F_2, .., G_4, F_2, .., G_3, cin, X, X_B, Y_B$ as well as the remaining X_q and Y_q are free in the two slices of the CLBs allocated to the register core. They can be used o implement another register in the same core.

If the core C_0 has no additional resource, I/O Multiplexing is commonly used to allocate the common resource C_0 either to C_1 or to C_2 depending on a predefined schedule. For two inputs I_1 and I_2 and a condition C, I/O multiplexing is implemented as follow: If $C = 1$ then $Z = I_1$, else $Z = I_2$. This is equivalent to the boolean equation $Z = C.I_1 + notZ = C.I_2$ which can easily be implemented in a 3-inputs LUT.
As we can see in this example, sharing resource in FPGA without additional overhead is possible, since the cores provided in the JBits environment use only a fraction of the resource allocated to them.

5. THE TEMPORAL PARTITIONING ALGORITHM

This section provides the description of our partitioning algorithm. The method *generate_partitions* (ALGORITHM I) is a list scheduling like method which takes as inputs a DFG G and a device type dev and return a list of all partitions *list_partition*.

At the begin all the nodes of the DFG are inserted in *list_nodes* in order of decreasing priority (line 2). In order to make a maximum use of overlapping capability, the priority is high in function of the non inclusion of the core in other cores, the non precedence by other cores in the list and the size of the core. At each step of the partitioning, a node *chosen_node* with maximum priority is removed from *list_nodes* and inserted in *allocated_list*(lines 6, 10 - 11, 14 - 16), which is the list of all elements of the current partition. It represents the resource running in the FPGA in a time interval. In order to make an efficient use of those resource for the next time interval, we first check for intersection between *chosen_node* and the nodes currently assigned to *allocated_list*. The function *get_best_overlapping_node* will return the node which has a maximum overlapping core with *chosen_node*. If no such node exists, *chosen_node* will be added to the current partition *allocated_list*, if the resulting partition fits in the device (lines 7 - 11). In the case where a best overlapping node exists, the overlapping concept as shown in section 4(lines 12 - 13) will be implemented. The current partition *allocated_list* is inserted in the list of all partitions when all the nodes in *list_nodes* have been processed (line 14). If the list of nodes *list_nodes* is not empty, a new partition is created and the processing continues (lines 3 - 14).

6. PLACEMENT OF MODULES IN CONFIGURATIONS

Having the list of all partitions, we use an enhanced *eigenvector based placement* to arrange the cores of each partition on the surface of the device and the resource inside the merged cores. This method has the advantage of placing Inputs and Outputs (I/O) elements at the boundary of the cores and the device, which makes pipelining easier. The method is based on the two dimensional spectral embedding of the nodes of the DFG using eigenvectors. Having a spectral embedding of the nodes in the plane, a post-processing method is used to successively remove the nodes not in the actual partition and fit the rest in the device area. The computation of the positions of modules on the device surface happen as follow: Given a DFG $G = (V, E)$ representing the modules and their interconnexion, the **connection matrix, degree matrix** and **laplacian** are first computed.

- The **connection matrix** of G is the symetric matrix $C = (c_{i,j})$ with $(1 \leq i, j \leq \mid V \mid)$ and $c_{i,j} = 1 \iff (v_i, v_j) \in E$.
- The **degree matrix** of G is the diagonal matrix $D = (d_{i,j})$, $(1 \leq i, j \leq \mid V \mid)$ with $i \neq j \to d_{i,j} = 0$ and $d_{i,i} = \sum_{j=1}^{|V|} c_{i,j}$.
- The **laplacian matrix** of G is the matrix $B = D - C$.

In [7] Hall proved that the r eigenvectors of the laplacian matrix related to the r smallest non zero eigenvalues of B define the coordinates of the $\mid V \mid$ modules of G in an r-dimensional vector space, such that the sum of the distances between the modules is minimal. We first compute the r smallest eigenvalues, then we use a post-processing step to generate the definitive position of each core in the corresponding bitstream.

The algorithm described here is implemented in a framework that we developed.

7. CONFIGURATION SEQUENCING

Given a DFG, our tool compute a **configuration graph**(fig.4) in which nodes represents configurations. Configurations communicate via **inter configuration registers**(fig.4), which are automatically inserted in bitstreams when an arc of the DFG connects two components in different configurations. The inter configuration registers are mapped in the CPU address room and are used for the communication between nodes in different configurations. The nodes of the configuration graph, that means the configurations are successively downloaded in the FPGA until the computation of the original DFG completes.

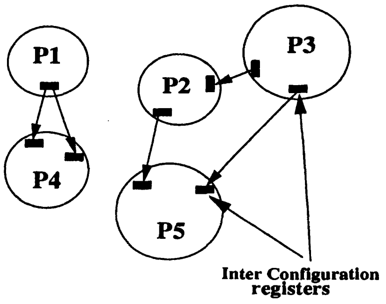


Figure 4. A Configuration Graph

8. CASE STUDY: NUMERICAL SOLUTION OF A DIFFERENTIAL EQUATION

To illustrate and show the efficiency of the method described in this paper, a numerical method for solving a differential equation as described in [12] is considered.

Fig 5 shows the DFG for solving a differential equation of the form $y'' + 3xy' + 3y = 0$ in the interval $[x_0, a]$ with step size d_x and initial values $y(x_0) = y_0$, $y'(x_0) = u_0$, using Euler's method as presented in [12]. We consider the worst case, where no additional resource is left in the available cores to implement the core overlapping. I/O multiplexing have to be considered in this case. The size of the different modules for the Xilinx Virtex architecture is given in table 1. As we can see, the size of a 2 inputs 1 output 16 Bit-Multiplexer is less than (10×1). Since the size of a 16 Bit-multiplier is more than (5×20), merging two cores sharing a multiplier module will have a reduction of the

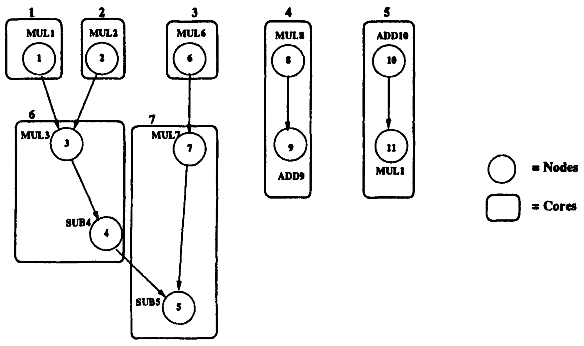


Figure 5. DFG For the DE-Integrator

Table 1. Cores Size(in CLB) for the Xilinx Virtex Architecture

Cores	X_Size	Y_Size
16-Bit Add	8	1
16-Bit Sub	8	1
16-Bit Comp	8	1
2 In/1-Out 16-Bit Mux	10	1
16-Bit Mul	5	20

size of a multiplier (5 × 20) and an augmentation of 3 times the size of a multiplexer(3 × (9 × 1)) on the size of the resulting core. The net gain we have is about 70 CLBs. With this observation the method described in 5 is suited to implement the DFG in a Virtex device. When targeting any of the Virtex above the Virtex 300 with a minimum area of (32 × 48), the design will completely fit in it. Only one partition will be generated and reconfiguration is not needed.

When targeting the Virtex 100 with size (20 × 30), the device can not hold more than 4 multipliers simultaneously. Another temporal partitioning algorithm will produce a minimum of two partitions in order to implement the DFG. For 1000 iterations of the DFG computations, the device will be configured 1000 times. With a reconfiguration time of $2s$ the time needed only for reconfiguration will be $2000s$. This solution is not applicable. The algorithm provided in this paper will generate only one partition for the same device. This means the reconfiguration time is reduced to zero.

The device will contain 3 multipliers, 2 substracters one adder one comparator and 9 multiplexers in the worst case. The total number of CLBs needed is 464. The design easily fits in the target device (Virtex 100). Table 2 gives an overview over the sharing of modules among cores in the device.

Table 2. Resource sharing for the problem of section 8

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Add1	-	-	-	-	X	-	-
Sub1	-	-	-	-	-	X	-
Sub2	-	-	-	-	-	-	X
Comp1	-	-	-	-	X	-	-
Mul1	-	-	-	X	-	X	-
Mul2	X	-	-	-	-	-	X
Mul3	-	X	X	-	-	-	-

9. CONCLUSION

This paper has presented a novel approach for the temporal placement of DFG on a reconfigurable platform. The paper shows that allowing overlapping of cores in the space and time can lead to a reduction in the number of partitions. Sharing modules in FPGAs and switching from one core to another can increase the latency of the partitions. Because the latency of a bitstream varies from nanoseconds to microseconds, while the reconfiguration time of an FPGA is more than a second, the overall performance of functions depends more on the reconfiguration time than the computation time.

References

[1] C. Bobda and N. Steenbock. Singular value decomposition on distributed reconfigurable systems. In *12th IEEE International Workshop On Rapid System Prototyping(RSP'01), Monterey California*. IEEE Computer Society, 2001.

[2] T. J. Callahan, P. Chong, A. Dehon, and J. Wawrzynek. Fast module mapping and placement for datapaths in fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 123 – 132, Monterey, California, 1998. ACM/SIGDA.

[3] J. M. P. Cardoso and H. C. Neto. An enhance static-list scheduling algorithm for temporal partitioning onto rpus. In *IFIP TC10 WG10.5 10 Int. Conf. on Very Large Scale Integration(VLSI'99)*, pages 485 – 496, Lisboa, Portugal, 1999. IFIP.

[4] D. Chang and M. Marek-Sadowska. Partitioning sequential circuits on dynamicaly reconfigurable fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 161 – 167, Monterey, California, 1998. ACM/SIGDA.

[5] Ejnioui and N. Ranganathan. Circuit scheduling on time-multiplexed fpgas.

[6] S. Guccione, D. Levi, and P. Sundararajan. Jbits: A java-based interface for reconfigurable computing, 1999.

[7] K. Hall. dimensional quadratic placement algorithm, 1970.

[8] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning tool for a class of dsp applications, 1998.

[9] H. Liu and D. F. Wong. Circuit partitioning for dynamicaly reconfigurable fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 187 – 194, Monterey, California, 1999. ACM/SIGDA.

[10] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures. In *IPPS/SPDP Workshops*, pages 31–36, 1998.

[11] A. Pandey and R. Vemuri. Combined temporal partitioning and scheduling for reconfigurable architectures. In J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, editors, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 93–103, Bellingham, WA, 1999. SPIE – The International Society for Optical Engineering.

[12] J. Teich, S. P. Fekete, and J. Schepers. Optimizing dynamic hardware reconfiguration. Technical Report 97.228, Angewante Mathematic Und Informatik Universität zu Köln, 1998.

[13] S. Trimberger. Circuit partitioning for dynamicaly reconfigurable fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 153 – 160, Monterey, California, 1999. ACM/SIGDA.

[14] M. Vasilko. Dynasty: A temporal floorplanning based cad framework for dynamicaly reconfigurable logic systems. In P. Lysaght and J. Irvine, editors, *Field Programmable Logic and Aplications FPL 1999*, pages 124–133, Glasgow, UK, 1999. Springer.

[15] M. Vasilko and G. Benyon-Tinker. Automatic temporal floorplanning with guaranteed solution feasibility. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 656–664, Villach, Austria, 2000. Springer.