Automated RTR Temporal Partitioning for Reconfigurable Embedded Real-Time System Design

C. Tanougast, Y. Berviller, P. Brunet and S. Weber
Laboratoire d’Instrumentation Electronique de Nancy, Université de Nancy 1, BP 239, Vandoeuvre Lès Nancy, France
{tanougast, berville, brunet, sweber}@lien.u-nancy.fr

Abstract

We present an automated temporal partitioning applied on the data-path part of an algorithm for the reconfigurable embedded system design. This temporal partitioning, included in a design space exploration methodology, uses trade-offs in time constraint, design size and FPGA device parameters (circuit speed, reconfiguration time). The originality of this partitioning is that it minimize the number of cells needed to implement the data-path of an application under a time constraint by taking into account the needs of bandwidth and memory size. This approach allows avoiding an oversizing of the implementation resources needed. This optimizing approach can be useful for the design of a dynamically reconfigurable embedded device or system. We illustrate our approach in the real time image processing field.

1. Introduction

The introduction of new high performance, high capacity field programmable gate arrays (FPGAs), combined with the emergence of hybrid and custom devices that combine FPGA fabrics with ASIC/full-custom components have made hardware reconfiguration a viable solution for flexibility in embedded systems. Indeed, the reconfiguration capability of SRAM-based FPGAs can be utilized to fit a large application onto an FPGA by partitioning the application over time into multiple parts. The objective is to swap different algorithms on the same hardware structure, by reconfiguring the FPGA in hardware several times in a constrained time and with a defined partitioning and scheduling [1]. The division into temporal parts is called temporal partitioning. Such temporally partitioned applications are also called Run-Time Reconfigured (RTR) systems.

Dynamic reconfiguration offers important benefits for the implementation of designs. Several architectures have been designed and have validated the dynamically reconfigurable computing concept for the real time processing [2-4]. However, the optimal decomposition (partitioning) of an algorithm by exploiting the run time reconfiguration (RTR) is a domain in which many works are left. Indeed, we observe that: Firstly, the efficiency obtained do not always lead to the minimal spatial resources. Secondly, the choice of the number of partitions is never specified. Thirdly, a judicious temporal partitioning can avoid an oversizing of the resources needed [5]. In this paper, we present an automatic RTR partitioning for the dynamically reconfigurable embedded real time system design in order to optimize the resources needed for a specific image processing application. This application field needs a high amount of computing resources. To overcome the effects of high oversizing of implementation resources for designing reconfigurable hardware, we demonstrated in [5] how an estimation of the number of partitions can be used as a pre processing step before temporally partitioning a design to increase the efficiency of the implementation. In the current work, we automate and extend our temporal partitioning technique to incorporate the memory bandwidth, the memory size, design space exploration techniques and demonstrate how this integrated processing can be used to optimize a temporally partitioned design.

In Section 2 we present the aim of our works. Section 3 presents our automated RTR temporal partitioning strategy. In Section 4 we illustrate, in an automatic way, the application of our partitioning in the real time image processing domain. In Section 5 we conclude and present future works.

2. Work focus

In constrast with others works [5, 6], we focus on the system design approach [7]. We try finding the minimal area that allows meeting the time-constraint. This is different from searching the minimal memory bandwidth or execution time which allows meeting the resources constraint. Here, we propose a temporal partitioning that uses dynamic reconfiguration of FPGA to minimize the implementation logic area. Each partition corresponds to a temporal floorplanning for dynamically reconfigurable logic embedded systems [8]. This is illustrated in Fig. 1.
We search the minimal floorplan area that implements successively a particular algorithm. This approach improves the performance and efficiency of the design implementation. Our aim is to obtain, from an algorithm description, a target technology and implementation constraints, the characteristics of the platform to design or to use. This allows avoiding an oversizing of implementation resources. For example, by summarizing the sparse information found in some articles [9-11], we can assume the following. Suppose we have to implement a design requiring $P$ equivalent gates and taking an area $S_{FC}$ of silicon in the case of a full custom ASIC design. Then we will need about $10 \times S_{FC}$ in the case of a standard cell ASIC approach and about $100 \times S_{FC}$ if we decide to use an FPGA. But the large advantage of the FPGA is, of course, its great flexibility and the speed of the associated design flow. This is probably the main reason to include a FPGA array on System on Chip (SoC) platforms. Suppose that a design is requiring that 10 % of the gates must be implemented as full custom, 80 % as standard cell ASIC and 10 % in FPGA cells. By roughly estimating the areas, we come to the following results: The FPGA array will require more than 55 % of the die area, the standard cell part more than 44 % and the full custom part less than 1 %. In such a case it could make sense to try to reduce the equivalent gate count needed to implement the FPGA part of the application. This is interesting because the regularity of the FPGA part of the mask of the SoC leads to a quite easy modularity of the platform with respect to this parameter.

Here, our goal is the definition of an automatic RTR temporal partitioning methodology, included in the architectural design flow, which allows minimizing the FPGA resources needed for the implementation of a time constrained images processing algorithm. This permits to enhance the silicon efficiency by reducing the reconfigurable array’s area [12] (optimize implementation area of designs). The challenge is obtaining computer-aided design techniques for optimal synthesis which include the dynamic reconfiguration in capability an implementation.

3. Automatic RTR temporal partitioning

3.1. Number and boundaries of partitions

Like the architectural synthesis [13], our approach is based on the elementary arithmetic and logic operators level of the algorithm (adders, subtractors, multiplexers, registers etc.). The analysis of the operators leads to a register transfer level (RTL) decomposition of the algorithm. That is why we assume the input specification of the application to be an acyclic data flow graph (DFG) where each node consists of an arithmetic and logic operation and the directed edges represent the data dependencies between operations [7]. The exclusion of cyclic DFG application is motivated by the following reasons:

- We assume that a co-design pre-partitioning step allows to separate the purely data-path part (for the reconfigurable logic array ) from the cyclic control part (for the CPU). In this case, only the data-path will be processed by our RTR partitioning method.
- In the case of small feedback loops (such as for IIR filters) the partitioning must keep the entire loop in the same partition.

Then, we search a trade-off between flexibility and efficiency in the programmable logic-array. The typical search sequence of temporal partitions in the design flow includes the following steps:

1) Definition of the constraints: the type of the design (use of a fixed-resources platform or target design) : time constraint, data-block size, bandwidth bottleneck, memory size, consumption) and the target technology.
2) DFG capture using design entry.
3) Determination of temporal partitioning.

4) Generation of the configuration bitstream of each temporal floorplan for the final design.

Here, we are only interested in dynamically configurable target design. In this case, the size of the logic array is tailored to implement the different temporal parts of an algorithm and the reconfiguration and memory control logic. In our case, the temporal partitioning methodology in the design flow is depicted on Fig. 2. Our method, which determines the minimal floorplan area, is structured on three parts.

\[
\text{Design capture} \quad \text{and annotation} \\
\quad \quad \text{Data-Flow Graph Description} \\
\quad \quad \text{Proto algorithm (pipeline)} \\
\quad \quad \text{Execution sequence} \\
\quad \quad \text{Scheduling sequence} \\
\quad \quad \text{Implementation (place & route)} \\
\quad \quad \text{Synthesis of temporal partitioning} \\
\quad \quad \text{Generate bitstreams for all configuration}
\]

**Figure 2. General outline of the temporal partitioning in the design flow.**

In the first part, we compute an approximation of the number of partitions (blocks A, B, C and D in Fig. 2). This part uses a target technology library that is based on a library of patterns for all possible data-path operators and constraint parameters. In the second part we deduce the boundaries of each temporal floorplan (block E). Finally we refine when it is possible the final partitioning (blocks E, F). The mapping and placement is carried out with the CAD tools of the target technology. Constraint parameters are used to adjust the temporal partitioning solution until all design constraints are met. This method can be seen as a heuristic approach.

From the formulation problem of the RTR partitioning for the real time application as a time constrained problem with dynamic resource allocations, we have demonstrated how to obtain the minimum number of partitions \( n \) and the corresponding optimal size \( C_n \) (number of cells) of each partition (see [7] for more details). These parameters correspond respectively to the number of sub graphs of the DFG that allows to execute the algorithm by meeting the time constraint \( T \) and the minimal amount of FPGA cells. The pseudo algorithm of the determination of the \( n \) and \( C_n \) is given below. We annotate the DFG by using the operator library. This library is target dependent and associates three attributes to each node of the graph G. These attributes are the bitwidth (.bit), the maximal path delay (.rt) and the number of elementary FPGA cells needed for an operator (.Area).

\[
V, N, T <= \text{constants} \quad \text{//Capture constant parameters of target and constraints} \\
G <= \text{DFG} \quad \text{// DFG capture of the application} \\
C <= 0 \quad \text{// Total area variable,} \\
T0 <= 0 \quad \text{//Maximal operator time variable}
\]

for each node \( Ni \) in \( G \)

\[
\text{TO} <= \text{max} (\text{TO}, \text{Ni.t}) \quad // \text{return current max of execution time} \\
C <= C + \text{Ni.Area} \quad // \text{add area of current node}
\]

end for

\[
n <= T / \left[ (N \cdot T0) + \text{rt()} \right] \quad // \text{compute n and C_n} \\
C_n <= C / n
\]

\( n \) is the number of reconfigurations \((n \in \mathbb{N})\). This is the result of the block D. This estimation corresponds to the upper limit of the processing time (time constraint in seconds called \( T \)) of one data block.

\( TO \) is the propagation delay of the slowest operator among all operators in the data-flow graph. This time equates to the maximum time between two successive nodes of the graph \( G \) thanks to the fully pipelined process (pipeline registers between all nodes of the graph).

\( N \) is the number of data in a block to process. We have assumed that the data to process are grouped in blocks of \( N \) data and the number of operations to apply to each data in a block is deterministic (i.e. not data dependent). A limit in the use of dynamic reconfiguration of FPGA has been exhibited [3], because the impact of reconfiguration time depends on the size of the data block to treat. The data must be computed in blocks with a significant size to reduce the reconfiguration overhead. We have neglected the impact of the latency time in comparison with the processing time. Indeed, in most application domain like image processing \( N \) is very greater than the number of pipeline stages that corresponds to the prologue and epilogue of the pipeline in each partition.

\( rt() \) is the reconfiguration time. In the case of a partially reconfigurable FPGA technology \( rt() \) can be approximated by a linear function of the area of the functional units being downloaded. The expression of \( rt() \) is equal to the number of cells required to implement one partition divided by the configuration speed. We consider that each reconfiguration overwrites the previous partition (we configure a number of cells equal to the size of the biggest partition). This guarantees that the previous configuration will never interfere with the current configuration. In the case of a fully reconfigurable FPGA technology the \( rt() \) function is a constant depending on
the size of the FPGA. In this case \( rt( ) \) is a discrete linear function increasing in steps corresponding to the different sized FPGAs.

The optimal size of the logic array is equal to the number of cells required to implement the entire Data-Flow Graph \( (C) \) divided by \( n \). The number of FPGA cells used is the maximal area among the partitions.

From the analysis of the value of \( n \), we can extract some information on the parameters of the system needed to realize the implementation. We describe these different cases below:

1) \( n > 1 \): It is possible to realize a RTR partitioning. In this case, an optimization is possible and allows obtaining a reduction of the logic area of the implementation with the technology used. \( n \) corresponds to the number of partitions that we are sure to obtain.

2) \( n \leq 1 \): In this case, with the RTR partitioning it is not possible to ensure the time constraint. Here, only static implementation with or without parallel processing allows respecting the time constraint.

If \( n = 1 \): Only static implementation is sufficient to meet the constraints with the used technology.

If \( n < 1 \): To ensure the constraint it is necessary to realize a static implementation with a processing parallelism. \( 1/n \) gives the degree of processing parallelism. In practice \( 1/n \in \mathbb{N} \). In this case, it is necessary to know if the target technology is interesting for the implementation of the application.

The pseudo algorithm of the partitioning scheme in Initial partitioning is given below. We consider a \( \text{First}_\text{Leaf}() \) function that takes a DFG as argument and which returns a terminal node. This function needs the operator library.

\[
\begin{align*}
G & \leftarrow \text{data flow graph of the application} \\
P_1, P_2, \ldots, P_n & \leftarrow \text{empty partitions} \\
\text{for } i \in \{1..n\} & \\
\quad C & \leftarrow 0 \\
\quad \text{while } C \leq C_n & \\
\quad & \text{append } (P_i, \text{First}_\text{Leaf}(G)) \\
\quad & C \leftarrow C + \text{First}_\text{Leaf}(G).\text{Area} \\
\quad & \text{remove } (G, \text{First}_\text{Leaf}(G)) \\
\text{end while} \\
\end{align*}
\]

We cover again the graph from the leaves to the root(s) by accumulating the sizes of the covered nodes until the sum is as close as possible to \( C_n \). These covered vertices make the first partition. We remove the corresponding nodes from the graph and we iterate the covering until the remaining graph is empty. The partitioning is then finished. There is a great degree of freedom in the implementation of the \( \text{First}_\text{Leaf}() \) function, because there are usually many leaves in a DFG. The only strong constraint is that the choice must be made in order to guarantee the data dependencies across the whole partition. The reading of the leaves of the DFG can be random or ordered. In our case it is ordered. We consider \( G \) as a two-dimensional table containing parameters relating to the operators of the DFG. \( \text{First}_\text{Leaf}() \) is carried out in the reading order of the table containing the operator arguments of the DFG (left to right) (see Fig. 3). In our case, the pseudo algorithm of \( \text{First}_\text{Leaf}() \) function is given below.

\[
\begin{align*}
M[i][j] & \leftarrow G \ ; \ \text{DFG capture} \\
\text{for } M[i][j] \text{ in } M & \\
\quad \text{if } M[i][j] \neq 0 & \\
\quad & \text{first}_\text{leaf}(G).\text{area} \leftarrow M[i][j].\text{area} \\
\quad & \text{first}_\text{leaf}(G).\text{tk} \leftarrow M[i][j].\text{tk} \\
\text{end if} \\
\text{end for} \\
\end{align*}
\]

**Figure 3. Labelling and reading of the DFG.**

3.2. Refinement after implementation

After placement and routing of each partition that was obtained in the initial phase we are able to compute the exact processing time. It is also possible to take into account the value of the synthesized frequency close to the maximal processing frequency for each partition. The analysis of the gap between the total processing time (configuration and execution) and the time constraint permits to make a decision about the partitioning. If it is necessary to reduce the number of partitions or possible to increase it, we return to the previous step described in Sub-section 3.1. with a new value for \( n \). Else the partitioning is considered as an optimal one (see Fig. 2). In this case, if time remains after the execution of all partitions, but it is not enough to add one, we decrease this time by reducing the working frequency in one or more partitions. Thus, we limit the consumption of the

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
application.

3.3. Optimizing memory bandwidth

The first aim of our methodology is to create partitions with areas as uniform as possible [7]. The second goal is to minimize the memory bandwidth. If this is the wish of a designer, he must also avoid going over technology constraints like memory bandwidth. To help the designer, we provide the data size requirement for each level of the DFG. By the way, we are able to see where the partitioning might produce a failure or an optimal solution. Moreover, in order to automate this refinement, the program can defer or anticipate the partitioning of the DFG to cut it where the memory bandwidth requirement is minimal. The search is restricted to a neighborhood of the theoretical partitioning points. This neighborhood is adjustable to keep the trade-off between uniform areas and memory bandwidth minimization. The size of the memory bandwidth (data size and real working frequency) along the DFG is known between each operator lines. When we create the DFG, edges receive a size argument that represents the data size of the bus between two nodes (operators). So, when we add all edges size arguments crossing nodes lines, we obtain the total data size in use at this point of the DFG. The pseudo algorithm of the partitioning refinement is given below. This code is put in the second pseudo code after the “While” loop. \( \alpha \) corresponds to the allowed relative variation that keeps the uniformity of the partitioning.

\[
\begin{align*}
\text{Pnode} &= \text{previous}_\text{First}_\text{leave}(G) \\
\text{Cnode} &= \text{First}_\text{leave}(G) \\
\text{Nnode} &= \text{Next}_\text{First}_\text{leave}(G) \\
\text{If} \quad [\text{Pnode}..\text{bit} = \text{Min} (\text{Pnode}..\text{bit}, \text{Cnode}..\text{bit}, \text{Nnode}..\text{bit})] \quad \text{and} \quad [(\text{C} - \text{Pnode}..\text{area}) > \text{Cn} (1 - \alpha)] \\
\text{Remove} (\text{Pnode}, \text{Pnode}) \\
\text{If} \quad [\text{Nnode}..\text{bit} = \text{Min} (\text{Pnode}..\text{bit}, \text{Cnode}..\text{bit}, \text{Nnode}..\text{bit})] \quad \text{and} \quad [(\text{C} + \text{Nnode}..\text{area}) < \text{Cn} (1 + \alpha)] \\
\text{Append} (\text{Pnode}, \text{Nnode})
\end{align*}
\]

3.4. Dynamic configuration and memory controller

A controller can be realized to manage the temporal partitioning procedure. This controller can read/write data in local memories and allows loading the configuration of the next temporal floorplan in the FPGA. In our case, this controller is a static design module. This controller is mapped in the first partition and it remains during all the processing steps. The use of partial reconfiguration allows keeping the controller in place while it manages the execution of the application in the remaining space of the FPGA.

The controller needs two counters to address the memories (size of \( \log_2 N \) for a N data block) , a state machine for the control of the RTR and the management of the memories for read or write access and a register (size in bits \( \log_2 n \)) that stores the partition number (n) found in the previous step of our method. This controller is illustrated in Fig. 4.

![Figure 4. General view of configuration and memories controller.](image-url)

4. Experimental results

In this Section, we illustrate our automatic partitioning in the image processing field. This application area is a good choice for our approach because the processings are generally characterized by regular operators (data-path) and the data are naturally organized in blocks (the images). Large blocks of data to process leads to a reduced overhead of reconfiguration time. In the image processing field the time constraint is usually the image acquisition period. We assume that the images are taken at a rate of 25 per second with a spatial resolution of 512² pixels and each pixel grey level is an eight bit value. The time constraint is equal to 40 ms for each image (definition of the constraint parameters: block B).

We performed an automated temporal partitioning on an estimator of the edges motion. This data-path algorithm is frequently used in an apparent motion estimator. It is composed of gaussian and averaging filters, followed by temporal and spatial derivatives and arithmetic divider.

We annotate the Data flow Graph. The characteristics (area, execution time) of all the operators in the DFG of the application are obtained from the FPGA technology target library [14]. We have choosen the Atmel AT40k FPGA-array family as the technology target (operator library and configuration speed). This family allows a partial reconfiguration of the logic array. Each configuration time depends on the quantity of logic cells used for each partition. The analyse of target datasheet leads a configuration speed of about 1365 cells per millisecond [7, 14]. The characteristics for our algorithm example are given in Table 1. It is the result of blocks A and C in the Fig. 2. The same considerations will apply to others FPGA target device technologies or systems.
After this annotation step, we compute the number of partitions needed to implement the algorithm in an optimized way (first pseudo code). For our algorithm example, among all operators (see Table 1), we can see that the slowest operator is a fifteen bit subtractor (operator execution time of 44.3 ns) and that we have to reconfigure 896 cells for a global implementation of the data-path part. From the time constraint, the data block size to process, and FPGA array family (reconfiguration speed and library characterising the low level operators), we obtain a value of four for \( n \) after refined of the partitioning (see Fig. 2). We deduce that the size of each partition \((C_i)\) should be about 216 cells. i.e. here, it is possible to implement this global data-path with about 216 cells in each partition with a reconfiguration time of 164 \( \mu \text{s} \). It is the result of block D. Then, we realize a partitioning which leads to partitions sizes as uniform as possible (216 cells) by executing the first leave function (second pseudo code).

We have tested the automatic partitioning on one ARDOISE\(^1\) module [3]. This module is constituted of one AT40K FPGA-array (equivalent of 20 K gates) and two 1MB SRAM local memory banks used as draft memory. Each local memory is connected to the FPGA-array with 32 data bits. The temporal partitions and the controller will be implemented in this FPGA. Partial results will be stored in the local memories while computing and reconfiguring. The FPGA-array receives the input pixels from one of its local memories, process and writes them into the other local memory. Next, after a new configuration it reads and processes the data from the last memory and writes results in the other local memory. The duration of this procedure is the duration of the image processing which should not exceed the frame period. Our method is not aimed to target such architectures with resources constraint. Nevertheless, the results obtained in terms of used resources and working frequency are still valid for any AT40k array.

Table 2 summaries the implementation results. We notice that a dynamic execution in four steps can be achieved in real time. We obtain a total processing time of about 38 ms. The homogeneity of partitions sizes is not really met because the partitioning takes into account the minimal memory bandwidth. The maximal number of cells by partition allows to determine the functional density gain factor obtained by the run-time reconfiguration implementation [12]. In this example, the functional density is near to 3 with respect to the global implementation of this data-path (static implementation) for real time processing without taking into account the resources of the controller for static or dynamic implementation.

In our applications we needed a maximum bandwidth of 30 bits. Ardoise module provides a local bandwidth of 32 bits that is sufficient to carry out the theoretical partitioning. Nevertheless we have to adopt a final partitioning that has the lowest bandwidth and that preserves the homogeneity of the resources in each partition. In order to know the needed resources, it is necessary to take into account the memory size needed to store the intermediate results and to quantify the cost of reconfiguration in terms of control and memory management (read and write counters (pointers), the configuration controller and the small associated state machine). The cost of the control part is following. The memory resources must be able to store two images (we assume a constant flow processing): memory size of 256 KB. In our case, the controller consists in two 18 bit counters (images of 512\(^2\) pixels), a state machine with five states, a 4 bit register to capture the number of partitions (we assume a number of reconfiguration lower than 16), a counter indicating the number of partitions, a 4 bit comparator and a toggle flip-flop to indicate to which alternate buffer memory we have to read and write. With the targeted FPGA structure, the logic area of the controller in each configuration stage requires a number of resources lower than 50 logical cells. Finally, the needed resources characteristics of the platform to use or to design for the example algorithm are: Memory size: less than 2MB (2 x 256 KB x 30), reconfigurable logic area (computing area + controller area): 304 cells, memory bandwidth: 30 bits with a minimal read/write frequency of 25.8 MHz.

5. Conclusion and future works

We proposed a automatic method for the RTR temporal partitioning of a dataflow graph that permits to minimize the array size of a FPGA by using the dynamic reconfiguration feature. This approach increases the silicon efficiency (enhances the functional capacity) by processing at the maximally allowed frequency on the smallest area and which satisfies the real time constraint. The method is based, among other steps, on an estimation of the number and boundaries of possible partitions by use of a characterized (speed and area) library of operators for the target FPGA. This leads to an optimized partitioning. Our method takes into account the memory bandwidth and memory size for the storage of the intermediate results. This approach allows obtaining the main parameters characterizing the architecture model, which implements a particular algorithm from the constraints. This is very interesting for development and optimization of reconfigurable embedded design.

\(^1\) This project involves ten French research labs which include our laboratory and is supported by the French agency for education, research and technology.
Table 1. Annotation and characteristics of the whole operators of the algorithm example

<table>
<thead>
<tr>
<th>Operators</th>
<th>DFG Annotation</th>
<th>Operators Characterization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quantity</td>
<td>Size (bits)</td>
</tr>
<tr>
<td>Absolute value</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Adder</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>Multiplication by 2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Multiplication by 128</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Subtractor</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>Division by 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Division by 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Register (pipeline or delay)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 2. Implementation results of our partitioning with an AT40K array

<table>
<thead>
<tr>
<th>Partition</th>
<th>Instantiate operator</th>
<th>Total number of Cells</th>
<th>Operator execution time (ns)</th>
<th>Total reconfiguration time (µs)</th>
<th>Partition processing time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5 Adders</td>
<td>225</td>
<td>27.1</td>
<td>173</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>3 Multiplication by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Subtractors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Absolute values</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Division by 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Division by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23 Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3 Subtractors</td>
<td>241</td>
<td>38.7</td>
<td>180</td>
<td>10.15</td>
</tr>
<tr>
<td></td>
<td>1 Multiplication by 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Division by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Multiplexer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14 Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2 Subtractors</td>
<td>248</td>
<td>38.7</td>
<td>180</td>
<td>10.15</td>
</tr>
<tr>
<td></td>
<td>2 Division by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Multiplexer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14 Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3 Subtractors</td>
<td>294</td>
<td>37.8</td>
<td>190</td>
<td>9.91</td>
</tr>
<tr>
<td></td>
<td>3 Division by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Multiplexer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21 Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This methodology for the RTR implementation of a design takes into account the features imposed by the target technology and allows avoiding an oversizing of the resources needed. We have illustrated the method by applying it on an image processing algorithm and by real implementation on a FPGA target technology. Moreover, this approach can be used on a platform with insufficient fixed-size FPGA resources because it allows evaluating the resources needed for an implementation. When the size of the FPGA is unknown, our method avoids the designer to run several times an area-constrained algorithm to try on which FPGA it can fit the algorithm.
At this time, we started to automate the partition search procedure, called DAGARD which means, in French, automatic partitioning of DFG for dynamically configurable systems, which is roughly a graph covering function. Currently, only the partition estimation part and boundaries generation between each partitions is automated. Our current perspective is an automatic synthesis of memory and configuration controller. Indeed, from the partitions computation, it is easy to automatically generate a fixed-resources controller for a particular implementation. In the future, it is necessary to take into account the power consumption that is often an important parameter for embedded systems design. This estimation depends on the target technology and the working environment like working frequency, resources needed for each partitions.

References


