Automating Production of Run-Time Reconfigurable Designs

Nabeel Shirazi and Wayne Luk
Department of Computing
Imperial College
180 Queen's Gate
London, England SW7 2BZ

Peter Y.K. Cheung
Department of Electrical Engineering
Imperial College
Exhibition Road
London, England SW7 2BT

Abstract

This paper describes a method that automates a key step in producing run-time reconfigurable designs: the identification and mapping of reconfigurable regions. In this method, two successive circuit configurations are matched to locate the components common to them, so that reconfiguration time can be minimized. The circuit configurations are represented as a weighted bipartite graph, to which an efficient matching algorithm is applied. Our method, which supports hierarchical and library-based design, is device-independent and has been tested using Xilinx 6200 FPGAs. A number of examples in arithmetic, pattern matching and image processing are selected to illustrate our approach.

1 Introduction

Hardware designers are used to develop circuits over space. Reconfigurable devices, such as SRAM-based FPGAs, provide an additional dimension: circuits can be spread over time as well. This flexibility enables a new design style of 'hardware-on-demand' or 'just-in-time processing' where circuit configurations are only loaded in the device when required. Such implementations are usually described as run-time reconfigurable (RTR).

While the advantages of reconfigurability are increasingly recognized, incorporating reconfigurable components into designs has yet to become routine for most FPGA users. Many found that, for instance, developing effective RTR designs is time-consuming and error-prone, and the design trade-offs involved are often unclear. This paper is intended to address such concerns by:

- presenting a method that automates the identification and mapping of reconfigurable components in RTR designs;
- illustrating the method using a number of examples in arithmetic and image processing;
- evaluating the design trade-offs involved in various reconfigurable implementations.

An outline of the paper is as follows. Section 2 provides an overview of a framework that we are developing. Section 3 covers an automatic method for extracting and mapping reconfigurable regions. Section 4 illustrates our method using a simple adder/subtractor example. Several more complex examples are discussed in Section 5, and concluding remarks are presented in Section 6.

2 Overview of Approach

The purpose of this paper is to describe a method for combining two or more designs into one reconfigurable design, based mainly on the identification of components common to these designs. The following explains how this method fits into a framework for developing RTR designs.

Previously we reported a model [4] and the associated development framework [6] for RTR designs. There are six steps in this framework: de-

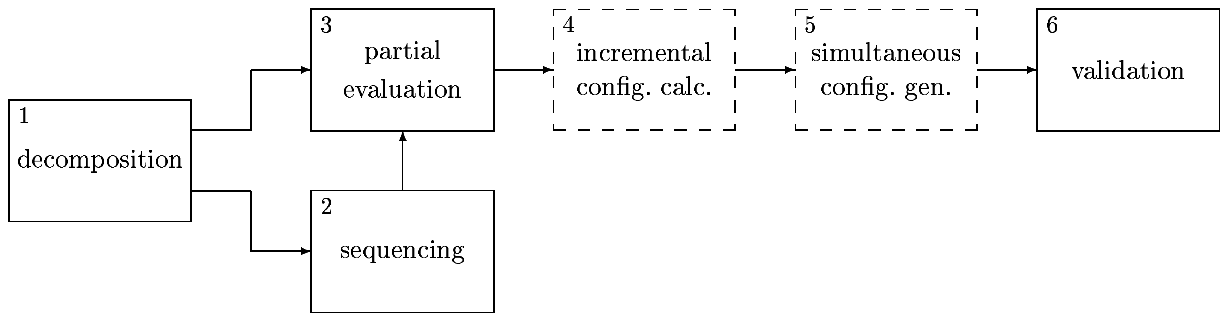

Figure 1 The six steps in our design framework [6]. The dotted boxes indicate that they are specific to devices or systems supporting partial reconfiguration or concurrent reconfiguration.

composition, sequencing, partial evaluation, incremental configuration calculation, simultaneous configuration generation, and validation (Figure 1). Prototype tools supporting the last five steps have been presented [6].

This paper focuses on the first step of the proposed development framework. The purpose of this step is to decompose a design into reconfigurable regions, each of which will be activated at the appropriate time during operation. It is a complex and difficult step. One challenge is to maximize resource usage while minimizing reconfiguration overhead, both in space and in time. Another challenge is to enable users to guide the selection of reconfiguration regions, while automating the low-level optimizations. To simplify this step, we divide it into two stages.

In the first stage, possible components for reconfiguration are identified, and a sequence of conditions for activating an appropriate component at a particular time is found. Most of the effort can be concentrated on selecting the top-level components in the design hierarchy for reconfiguration, and on producing the associated activation sequence.

In the second stage, successive configurations will be optimized to achieve the desired trade-offs in reconfiguration time, operation speed and design size. Components and connections common to two or more successive configurations will be identified and will not be reconfigured. Since the effectiveness of the second stage depends on the effectiveness of the first, several iterations between the two stages may be required to obtain an optimal decomposition.

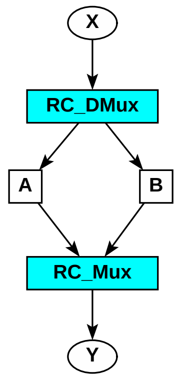

Figure 2 A static network modelling a design that can behave either as A or as B, depending on the control blocks RC_DMux and RC_Mux. The select input for these control blocks is not shown.

The above procedure can be explained using our model [4] for reconfigurable designs. In this model, a component that can be configured to behave either as A or as B is described by a network with A and B connected between two control blocks. The control blocks, RC_DMux and RC_Mux, route the data and results from the external ports x and y to either A or B at the desired instant (Figure 2). The select input for these control blocks is not shown. Each control block will be mapped either into a real multiplexer or demultiplexer to form a single-cycle reconfigurable design, or into virtual ones which model the control mechanisms for replacing one configuration by another [4]. Some examples of these two styles can be found in Section 5. In the first stage of the decomposition step described above, control blocks are placed between components close to the top of the design hierarchy. In the second stage, the design hierarchy is transformed such that the control blocks are moved towards the lower levels to reduce the size of reconfigurable regions. This technique, together with the optimizations in other steps in Figure 1, can be applied to minimize reconfiguration time.

The proposed method, which complements the low-level tools described previously [6], can be used in three different ways. First, it can be used to introduce reconfiguration in an existing circuit with placement information, since the matching procedure, presented in the next section, takes such information into account when it is available. Two designs can be combined into a new circuit, with compatible placement information, using RC_Mux and RC_DMux. Second, our method can still be used when placement information is not available; placement constraints will be added to the generated configurations for subsequent placement. Placement constraints can also be produced for designs based on relocatable parametrized libraries [3]. Finally, if placement information is available for some but not for all of the components in a design, our method can be used to generate appropriate placement constraints for components without them.

To facilitate designers to control the reconfiguration process at a relatively high level and to enable experiments on initial identification of reconfigurable regions and activation sequences, the first stage of the decomposition step has not been automated at the time of writing. In the following sections, we describe a method that principally automates the second stage: combining multiple designs to form a single RTR design, and generating its description with control blocks in a form suitable for further processing by other tools [6].

3 Producing Reconfigurable Components

In this section, we first present an overview of the matching procedure using a simple example. The principal steps will then be described in detail.

3.1 Overview

Given two successive circuit configurations, the matching procedure is to identify the components common to both configurations so that only the parts that are different need to be reconfigured. A combined circuit description is then produced with the reconfigurable components connected by RC_Mux and RC_DMux, as discussed in the previous section.

Our method contains three steps:

1. representing the components in the two circuit configurations as nodes in a bipartite graph – a graph whose vertices can be covered by two independent sets – with weighted edges indicating the strength of possible matches between components of the two configurations;

2. computing the best match for each node in one configuration with a node in the other configuration, taking into account the value of the weights;

3. inserting RC_Mux and RC_DMux to produce a combined circuit with explicit reconfigurable regions.

A simple example of possible matches is shown in Table 1, where configuration P has four components $p1, \ldots, p4$, and configuration Q has five components $q1, \ldots, q5$. Figure 3(a) contains the representation of Table 1 as a bipartite graph showing possible matches after weights have been assigned. The result of the graph after matching is shown in Figure 3(b).

Since designs are usually organized hierarchically, the weight assignment and matching steps are carried out recursively down the levels of the

Configuration P	Configuration Q
$p1$	$q1, q4$
$p2$	$q1$
$p3$	$q3, q4$
$p4$	$q2, q5$

Table 1 Possible matches between components $p1, \ldots, p4$ in P and $q1, \ldots, q5$ in Q.

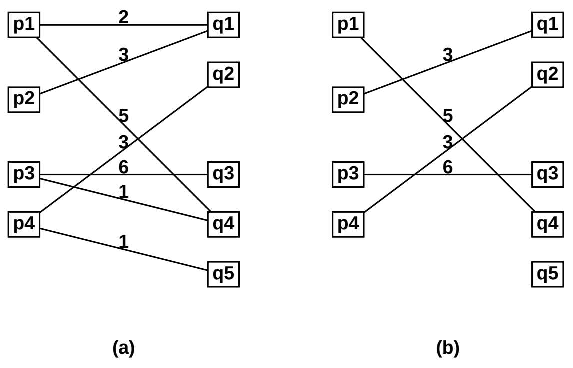

Figure 3 Weighted bipartite graphs for configurations P and Q before and after matching.

design hierarchy for components of different types but with the same ports. Since the hierarchy of two designs may not always be the same, the user may need to explore the extent to which the design should be flattened. We are currently exploring automatic techniques for determining how designs can be flattened.

In the following, we describe how a weighted bipartite graph can be built from two circuit configurations, and how the graph can be matched and how the resulting combined configuration is produced.

3.2 Graph Weighting

The netlists of the two circuits that are to be matched are processed to produce dataflow graphs. The internal nodes of these dataflow graphs make up the elements of two independent sets V_1 and V_2 for the bipartite graph. To indicate the strength of the correspondence between two components, three different criteria are used to produce weights for the edges of the bipartite graph: component type, position in the FPGA, and depth from matched ports.

The first and the most obvious criterion used in weighting the graph is the type of the two components. There are two possibilities: exact matches and similarity matches. For example, the structure of a ripple adder is similar to that of a ripple subtractor. A sample list of Xilinx 6200 library components [3] that have similar structure has been compiled. An exact match or a similarity match leads to the assignment of a weight between the two components belonging to different configurations. Two similar components will be labelled so that the matching can terminate at this level, provided that another tool, such as the *ConfigDiff* tool [6], will be able to extract the reconfigurable parts automatically.

Position information, if available, can reinforce the weights when the component types are identical or similar. A position match occurs if the locations of two components are the same in the FPGA. Position information can be a relative location in a level of hierarchy, or an absolute location depending where in the netlist hierarchy the location attribute is given.

Weighting components by type and position does not take their connectivity into consideration. To deal with this important aspect, we need to find the corresponding ports for the two dataflow graphs to be matched. We can then calculate the *depth* of a node from an input or an output port, which is the number of nodes reachable from that port to this node, and the maximum depth will be recorded. To deal with cyclic dataflow graphs, we first find the strongly-connected components, which correspond to feedback loops. The depth calculation for a strongly-connected node will terminate if a component is encountered twice. Nodes in the two sets with the same depth from two matched ports will be assigned a weight value, which will be used later in the matching algorithm. The weights for nodes of the same depth will be given a relatively lower value than component type weights, since multiple components may have the same depth.

The graph weighting procedure will be illustrated by an adder/subtractor example in Section 4.2.

3.3 Graph Matching and Combination

Once the edges of the bipartite graph are weighted, the matching can be performed. Our graph matching algorithm is adapted from a well-known algorithm proposed by Hopcroft and Karp [2]. It is chosen because it runs in $O((m+n)\sqrt{n})$ computational steps, where m is the number of edges and n is the number of nodes in the bipartite graph. The details of this algorithm have been covered in many textbooks [7], and will not be described here. In our implementation, we keep a list of components which have already been matched to avoid unnecessary re-matching.

Once the matching has been performed, the two circuits are combined by including RC_DMux and RC_Mux components [6] following the rules below.

- If two matched components are of the same type, only the component from the set V_1 is added.

- If two matched components are of different types, then both components are added and are connected in parallel to an RC_DMux and an RC_Mux, forming a reconfigurable region. If the positions of the two components are different, a procedure for determining the optimal positions will be invoked.

- If there are remaining unmatched components from either graph, they will be paired with a wiring component from the other graph and both are then combined to form the new graph as in the previous step.

The graph matching and combination procedure will be illustrated by an adder/subtractor example in Section 4.3.

4 Adder/Subtractor Example

In a Xilinx 6200 FPGA [1], an n-bit ripple adder/subtractor using only localized routing can be implemented using $6n$ components. The size of this adder/subtractor can be reduced by 33%, if the adder is changed into a subtractor using run-time reconfiguration [6].

This design can be implemented by inverting one of the input bits of each full adder component, and also changing the carry-in to the adder array from a logic zero to a logic one. The repeating unit of the ripple adder/subtractor consists of a two by two array of components, and the gate that inverts one of the inputs is located at every other component. The schematic for the repeating component is shown in Figure 4. The inversion can be performed by reconfiguring the XOR gate at u1 to become an XNOR gate.

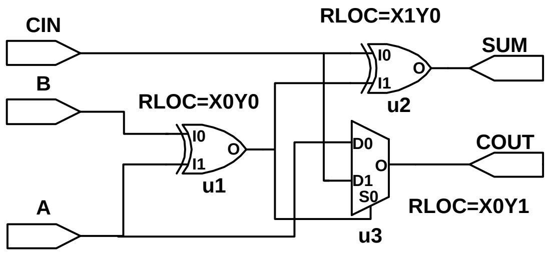

Figure 4 A full adder circuit. A full subtractor circuit can be obtained by replacing the XOR gate at u1 by an XNOR gate, with the same position attribute RLOC=X0Y0.

4.1 Overview

To illustrate the graph matching procedure, we apply our algorithm to the repeating components of the ripple adder/subtractor and automatically find that the reconfigurable region is the XOR/XNOR gate. The dataflow graph (DFG) for the repeating component for the adder is shown in Figure 5. In the DFG, input/output ports are denoted by oval shaped nodes and internal components are denoted by rectangular boxes.

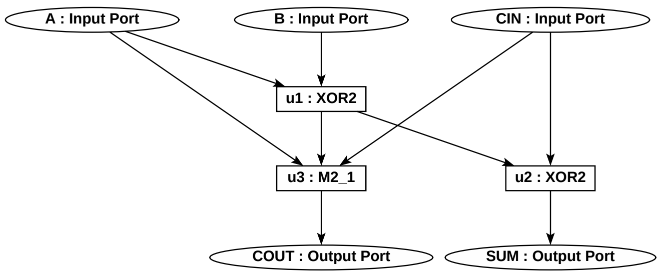

Figure 5 DFG representation of a full adder. The DFG of a full subtractor can be obtained by replacing the node u1:XOR2 by u1:XNOR2.

Only the internal components from each circuit are added to the weighted bipartite graph shown in Figure 6. The three components from the full adder graph are captured by the set V_1, and are placed on the left-hand side of the bipartite graph. Similarly, components from the full subtractor are captured by the set V_2 and are placed on the right-hand side of the bipartite graph. The weights are added using the rules discussed in the previous section.

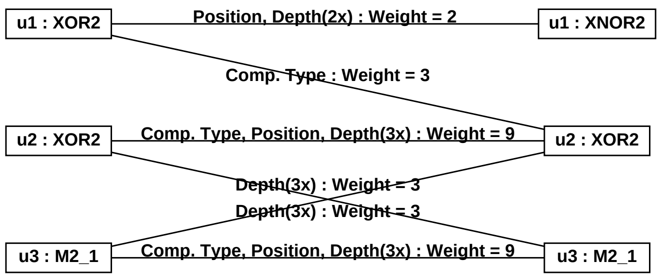

Figure 6 Weighted bipartite graph before matching. The nodes on the left are from the set V_1 for the adder, and those on the right are from V_2 for the subtractor. The type of match is indicated on each edge. Depth(nx) indicates that n depth matches have occured.

4.2 Graph Weighting

Let us illustrate the graph weighting procedure described in Section 3.2. Note that there are two XOR gates in the vertex set V_1, and only one XOR gate in V_2. Both XOR gates in V_1 produce a component type match with the XOR gate in V_2. The edge between u1:XOR2 of V_1 and u2:XOR2 of V_2 shows the component type match and is given a weight value of 3.

The gates u1:XOR2 in V_1 and u1:XNOR2 in V_2 are both placed at $x = 0$, $y = 0$ in the FPGA as shown by the RLOC attribute in Figure 4. Since the component types do not match, a position match weight has not been added to the edge between the two nodes. A position match has been obtained between nodes u2:XOR2 in V_1 and u2:XOR2 in V_2, since they are positioned at the same location $x = 1$ and $y = 0$ and are of the same component type. The position match is indicated on the edge between the two nodes, and a weight of 3 has been added.

A depth match is performed by first calculating depth information from each of the matched input ports. When port A is used as the starting point for the adder component, u1:XOR2 has a maximum depth of 1, and both u2:XOR2 and u3:M2_1 have the maximum depth of 2. Depth information from the other input ports is calculated as well. This information is used to match components reachable from the matched ports. For instance, port A of the adder is matched with port A of the subtractor. A depth match starting from this port results in one depth match between each of the five node pairs (Figure 7). The second depth match from port B results in five more depth matches between the nodes that are matched when port A is used. Finally the third depth match starting from the CIN port results in a match between only four components and not five, because node u1:XOR2 (Figure 5) cannot be reached from the CIN port in the adder graph, and similarly u1:XNOR2 cannot be reached from the CIN port in the subtractor circuit.

After the weights have been added to the edges, the edge between u2:XOR2 in V_1 and u2:XOR2 in V_2 has a weight of 9 (Figure 6). This is due to a weight of 3 from the component type match, 3 from the position match and 1 from each of the three depth matches.

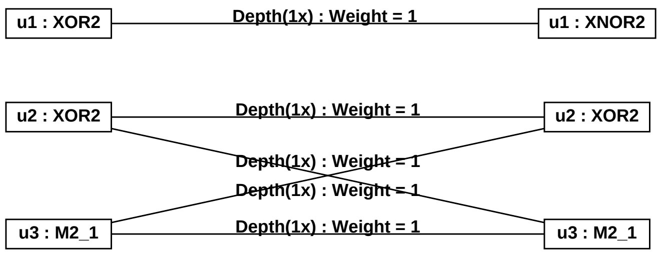

Figure 7 Weighted bipartite graph after depth matching for Port A.

4.3 Graph Matching and Combination

We can now illustrate the procedure for matching and combining graphs discussed in Section 3.3 using the adder/subtractor example. The matching is carried out by the Hopcroft/Karp algorithm, taking into account the value of the weights. For instance, since the gate u2:XOR2 in V_2 is weighted stronger to u2:XOR2 in V_1 than to u1:XOR2 in V_1, u2:XOR2 in V_2 is matched with u2:XOR2 in V_1. The resulting weighted bipartite graph after matching is shown in Figure 8. Note that the same graph can be obtained even if position information is not available for matching. Once we know which nodes are matched, we build the new DFG shown in Figure 9 using the method discussed in Section 3.3, which combines the repeating components from the adder and the subtractor.

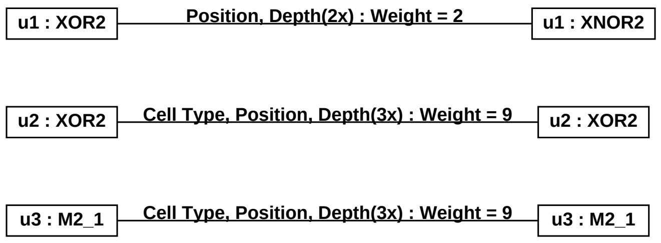
u1 : XOR2
Position, Depth(2x) : Weight = 2
u1 : XNOR2
u2 : XOR2
Cell Type, Position, Depth(3x) : Weight = 9
u2 : XOR2
u3 : M2_1
Cell Type, Position, Depth(3x) : Weight = 9
u3 : M2_1

Figure 8 Weighted bipartite graph after matching.

The reconfigurable region is delimited by the RC_DMux and the RC_Mux. The select value for the RC_DMux and the RC_Mux is given during the partial evaluation stage in our compilation tools [6], and either the XNOR or XOR gate is deleted along with the RC_Mux and the corresponding RC_DMux.

In this example we are able to identify the reconfigurable region automatically. We combine the adder and subtractor DFGs, so that the reconfigurable components can be selected either at run time or at compile time. Finally the components in the reconfigurable region are positioned so that reconfiguration time will be minimized.

5 Examples and Evaluation

Our previous work involves several hand-crafted RTR designs using the Xilinx 6200 FPGAs [4], [6]. Much effort went into finding the smallest reconfigurable region and the placement of the components in order to minimize reconfiguration time.

In the following, we evaluate our approach to automating RTR designs using several pattern matchers and image filters. The results (Table 2) from our tools are the same as the hand-crafted versions. For each application, two RTR designs are produced: a single-cycle reconfigurable (SCR) design, and a partial RTR design.

Example	Size (cells)	Speed (ns)	Reconfig. Cycles
Adder/Subtractor:			
SCR design	48	63	1
Partial RTR design	32	47	4
Pattern Matcher:			
SCR design	128	190	1
Partial RTR design	64	90	32
1-D Image Filters:			
SCR design	126	67	1
Partial RTR design	90	52	22

Table 2 Statistics for some applications.

In each of the examples, there is a reduction in size and an increase in operation speed when par-

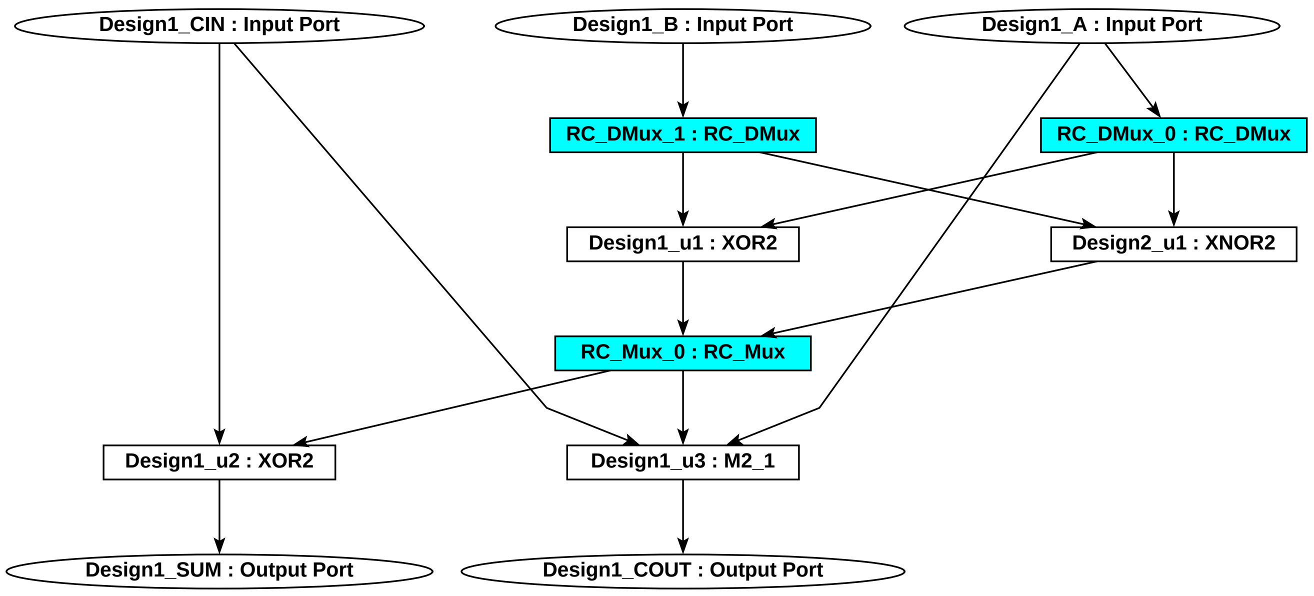

Figure 9 Resulting DFG by combining a full adder and a full subtractor. Note that the reconfigurable region is delimited by the RC_DMux and the RC_Mux.

tial run-time reconfiguration is carried out by the FPGA – these are the partial RTR designs in Table 2 which take multiple reconfiguration cycles. As long as the circuits are not reconfigured too often, this kind of partial run-time reconfiguration is beneficial. In each example, the reconfiguration time is less than $2\mu s$ if the partial configuration is loaded at the maximum configuration speed of the Xilinx 6200. The reconfiguration time is relatively low compared to the number of clock cycles the design would have to run in data processing mode to justify run-time reconfiguration. When appropriate, a "break-even" point can be calculated; for instance, the break-even point proposed by Wirthlin and Hutchings [8], based on their functional density metric, is less than 32 clock cycles for each of the examples in Table 2.

The single-cycle reconfigurable (SCR) designs are produced by combining two designs together automatically using the procedure described in Section 3, and the result is optimized further by hand. For example, an adder is combined with a subtractor by adding circuitry to calculate the two's complement of one of the inputs based on a control signal. The SCR design can be reconfigured in one clock cycle by changing the value of the control signal. The partial RTR designs are also produced by combining two designs, and then the compilation tools presented in [6] are used to produce the initial and incremental configurations that are loaded during run time.

5.1 Pattern Matcher Example

Our first example is a 32-bit pattern matcher (Figure 10). The partial RTR version of this design takes up $32 \times 2 = 64$ FPGA cells, whereas a design including an additional shift register for storing the pattern and an additional row of comparators will be twice as large [6]. By adding the register and comparator to the repeating cell of the pattern matcher, it also increases the propagation delay by 52%. In the worst case, if wildcarding cannot be used, it takes 32 configuration cycles to reconfigure the 32-bit pattern matcher, i.e. one cycle for each repeating cell.

The partial RTR design for the pattern matcher is generated automatically by combining two circuits that were designed specifically to generate a match for a particular constant value. If the individual bits of the constant value are different, an RC_Mux/RC_DMux pair is inserted to reconfigure between the gates used to cause a match at the particular bit. If the two circuits that are to be matched are designed to cause a match for the two extreme constant values, i.e., all 1's or all 0's, then each of the repeating cells can either match for a 1 or a 0, depending on the select value of the inserted RC_Mux/RC_DMux at the partial evaluation stage. By matching these two 'extreme' designs, not only can we match the all 1's or all 0's cases, but also any of the 2^n constants since the RC_Muxes/RC_DMuxes can be individually controlled.

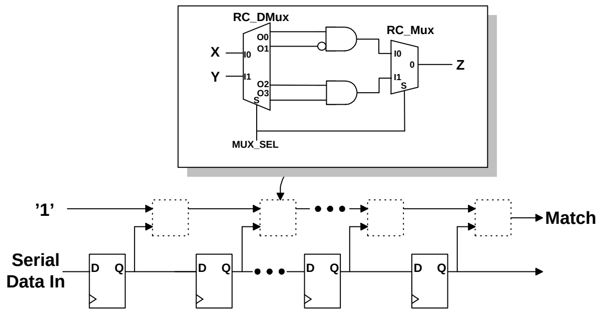

Figure 10 A multi-bit pattern matcher.

Furthermore, if the pattern matcher is pipelined by inserting a register in the carry chain between each stage, the computation and reconfiguration can be overlapped by using the "morphing" technique [5]. The pipelined circuit's propagation delay is reduced to $2.8ns$ from $90ns$ and the reconfiguration time is still 32 cycles if morphing is used. The size will also remain the same since a register and a gate can be combined in one cell in the Xilinx 6200 [1].

5.2 Image Filter Example

Our second example involves reconfiguration between two one-dimensional image filters. The first filter is a Sobel edge detector for locating vertical edges in an image, and the second filter is a Gaussian filter for removing noise in an image. The DFG for these filters are shown in Figure 11 and the functionality is described in more detail in [4].

This example shows the advantage of hierarchical matching instead of matching on a single level. While descending the design hierarchy to match the components Sub1 and Add2, it can be found that the reconfigurable region is just a single gate in the repeating unit in the new component Design1_Add2 : RC_Mux_Adder_Subtractor which is highlighted in Figure 12. Component Add1 is not matched and therefore reconfiguration occurs between the adder and a set of wires. Space is also allocated in the FPGA so that there is room for Add1 to be swapped without having to move the other components.

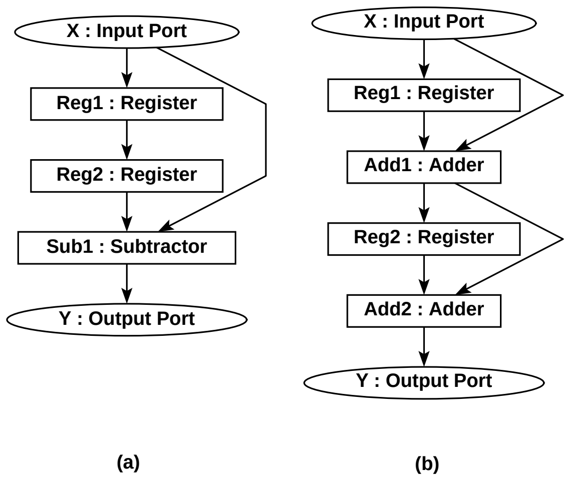

Figure 11 (a) 1-D Edge Detector, (b) 1-D Gaussian Filter.

The filters are set up so that when the result is read, a clock pulse would be generated and new data will be clocked into the filter. This involves a feedback circuit with the register containing the result. Our matching algorithm is able to handle feedback circuits, since the depth-matching procedure can deal with cyclic graphs (Section 3.2).

The partial RTR versions of the image filters are 29% smaller and 22% faster than the SCR design and the reconfiguration time is only 22 cycles.

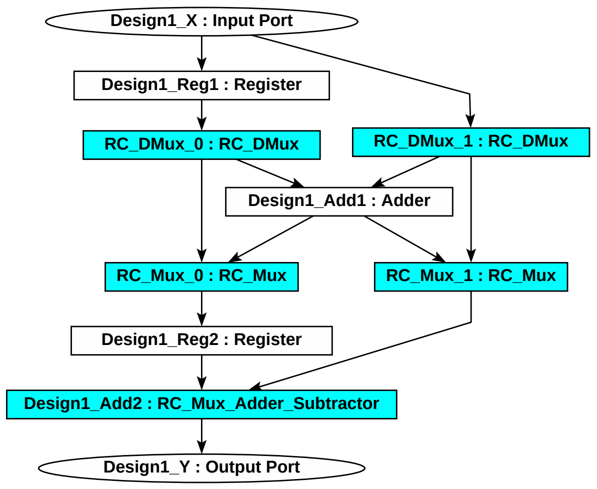

Figure 12 Combined Edge Detector and Gaussian Filter.

6 Concluding Remarks

This paper describes a method for automating the identification and mapping of reconfigurable regions for RTR designs. The method is based on an efficient algorithm for matching weighted bipartite graphs. Several examples for Xilinx 6200 FPGAs have been used to illustrate our approach. Current and future work includes refining the tools and applications, investigating the automation of the initial decomposition stage and the optimization for three or more sequential configurations, and exploring the adaptation of the proposed techniques for interactive design and run-time synthesis.

Acknowledgments

Many thanks to John Gray, Tom Kean, Mehdi Shirazi, Dimitrios Siganos, Markus Weinhardt, and Bill Wilkie for their comments and discussions. The support of UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658), Xilinx Development Corporation and a UK Overseas Research Student Award is gratefully acknowledged.

References

[1] S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap Processor Interface", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 36–43.

[2] J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ Algorithm For Maximum Matchings in Bipartite Graphs", in *SIAM Journal of Computing*, Vol. 2, No. 4, December 1973, pp. 225–231.

[3] W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A Framework for Developing Parametrised FPGA Libraries", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, R. Hartenstein and M. Glesner (eds.), LNCS 1142, Springer, 1996, pages 24–33.

[4] W. Luk, N. Shirazi and P.Y.K. Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems", in *Proc. FCCM96*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1996, pp. 167–176.

[5] W. Luk, N. Shirazi, S.R. Guo and P.Y.K. Cheung, "Pipeline Morphing and Virtual Pipelines", in *Field Programmable Logic and Applications*, W. Luk, P.Y.K. Cheung and M. Glesner (eds.), LNCS 1304, Springer, 1997, pp. 111–120.

[6] W. Luk, N. Shirazi and P.Y.K. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs", in *Proc. FCCM97*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1997, pp. 56–65.

[7] D.B. West, *Introduction to Graph Theory*, Prentice Hall, 1996.

[8] M.J. Wirthlin and B.L. Hutchings, "Improving Functional Density Through Run-Time Constant Propagation", in *FPGA'97*, ACM Press, 1997, pp. 86–92.