Run-time Generation of Partial Configurations for Arithmetic Expressions

Miguel L. Silva
DEEC, Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
Email: mlms@fe.up.pt

João Canas Ferreira
INESC Porto, Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
Email: jcf@fe.up.pt

***Abstract*— Adaptive embedded systems can achieve enhanced flexibility by performing run-time reconfiguration of hardware. This paper describes a method to generate at run-time new partial FPGA configurations corresponding to arithmetic expressions. This is achieved by merging available partial bitstreams of arithmetic components to produce a new partial bitstream for a specific FPGA area. The connections among the components are mapped to the switch matrices of the reconfigurable fabric, and the corresponding information is added to the new partial configuration. The proposed method was implemented for a Virtex-II Pro FPGA with a 300 MHz PowerPC 405 CPU. It was used to create partial configurations in less than 69 s for sets of arithmetic circuits with up to 25 components and 208 connections.**

I. Introduction

Embedded systems that are able to modify their behavior in response to changes in the environment or in the system's goals are gaining in importance [1], [2]. Dynamically reconfigurable FPGAs are a natural implementation platform for such systems, because they provide the basic capabilities required for hardware modification at run-time. Embedded systems are often resource-constrained, which makes adaptable hardware support very attractive, even for operations done in software in more powerful systems. By performing run-time reconfiguration (RTR), hardware resources can be exploited according to the immediate requirements of the executing application, instead of being committed permanently to a single task.

Normally, RTR uses partial bitstreams created at design time. Creation of configurations at run-time is justified when there are too many possibilities, or when they depend on information that only becomes available as the application runs.

Usually, synthesis tools must be run (at design time) to create each partial configuration. An alternative, simpler approach based on building a partial bitstream by combining bitstreams of smaller components is described by [3]. Since this approach does not rely on the synthesis of logic descriptions, it is a good candidate for use in run-time generation.

Configuration bitstreams target a specific FPGA area. If that area changes after creation, the bitstreams must be relocated to the new target area. This capability makes for more flexible system deployment, so several approaches to the relocation of partial bitstreams have been proposed [4], [5]. The same requirement for relocation arises when bitstreams are combined.

The run-time generation of configurations in embedded systems has been rarely addressed. A channel router for the Wires-on-Demand RTR framework is described in [6]. It uses a simplified resource database and simple algorithms to find local routes between blocks using relatively few computational resources. The possibility of running in an embedded system is mentioned, but no results are reported.

A more primitive version of the bitstream assembly approach used in this work is described in [7], where inter-module connections are selected from a table of predetermined routes. Although fast, the approach has limited flexibility.

Here we address the problem of generating at run-time partial configurations for sets of arithmetic expressions. We avoid the use of a predetermined set of routes, and include support for automatic placement of the components. The work assumes the presence of a CPU, and the capability of loading the partial bitstream to a specific FPGA area without disturbing the operation of other parts of the system. A reserved area of the FPGA should be set aside for hosting the RTR circuits. For each component, an abstract description and a partial bitstream must be available. The abstract description specifies the component's bounding-box, the position of the I/O terminals at its periphery, and the internal location of any special resources (e.g., dedicated multipliers).

The generation of partial configurations is, by necessity, closely linked to the underlying reconfigurable fabric. For our proof-of-concept implementation we used a Virtex-II Pro FPGA [8], which supports active partial reconfiguration, and has an internal access port for configuration.

The paper is organized as follows. Section II describes the abstract model of the reconfigurable infrastructure resources. Section III presents the main steps of run-time generation of partial configurations. The results obtained with a proof-of-concept implementation are described in section IV. Concluding remarks are presented in section V.

II. Resource Modeling

The creation of a new configuration starts with a directed acyclic graph (DAG) that describes the connections among

978-1-4244-7773-9/10/$26.00 ©2010 IEEE

the arithmetic components. If there are no common sub-expressions, the description becomes a set of trees (one for each arithmetic expression). The configuration generation process treats components as black boxes: they cannot overlap, and no interconnections can traverse them. In the present approach, components are grouped in vertical stripes. The position of a component inside a stripe and the width of the stripe depend on the physical resources used by the component. Only connections between components in adjacent stripes are allowed. This restriction simplifies generation by ensuring that the interconnections do not interfere with the rest of the system, and by reducing the search space. There may be empty CLB (Configurable Logic Block) columns between stripes. The current implementation allows for only one additional empty column to account for unused block RAMs.

The Virtex-II Pro FPGA has a segmented interconnection architecture, where individual segments are used to connect switch matrices [9]. Each switch matrix also connects to a CLB or dedicated block. From the large number of available routing resources, only a subset is used here:

- direct connections (vertical, horizontal and diagonal connections to neighboring CLBs);
- double lines (connections to every first and second CLB in all four directions);
- vertical hex lines (connections to every third or sixth CLB above or below).

Long lines (wires that distribute signals across the full device height and width) are not used, since they can interfere with circuitry outside of the reserved area. Similarly, horizontal hex lines would reach beyond the area between stripes. As a result, the model of the switch matrix contains 116 pins:

- 16 direct connections to the 8 neighboring CLBs;
- 40 double lines: 10 in each of the four directions up, down, left and right;
- 20 vertical hex lines: 10 upwards and 10 downwards;
- 8 connections to the outputs of the 4 slices in the associated CLB;
- 32 connections to the inputs of the 4 slices in the associated CLB.

III. Generation of Partial Configurations

The generation of a new partial configuration starts from a component netlist, where each component represents an arithmetic operator. The generation has two main stages: 1) assigning a location to each component; 2) creating the connections from output to input terminals (including the terminals on the interface of the reserved area).

The strategy for determining the location of a component is to find an arrangement of components in columns (stripes), so that directly connected components are adjacent to each other. The arrangement in columns was chosen because it matches the reconfiguration mechanism of Virtex-II-Pro FPGAs, where the smallest unit of reconfiguration data applies to an entire column of resources. This arrangement also fits naturally

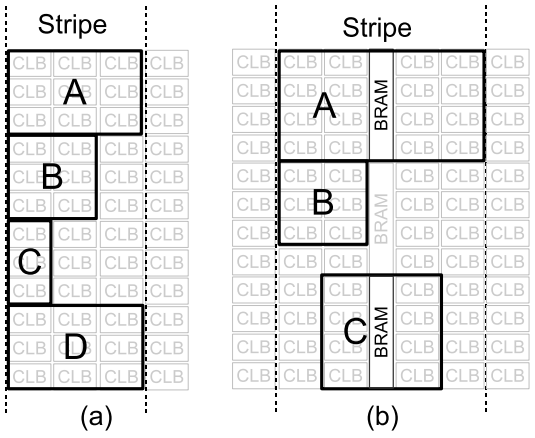

Fig. 1. Location of components in stripes. (a) Typical placement for components that only contain CLBs; (b) placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

the tree-like structure of many arithmetic expressions. Two examples of component placement inside a stripe are displayed in figure 1.

The first step of the placement algorithm is to group the components by topological level: the first level contains the components whose inputs are connected to the interface of the dynamic area; the second level contains that components that have all their input terminals connected to first-level components, and so forth. A component with more than one input source will be assigned to the level following its highest-numbered source.

Each level is examined in order to determine the set of contiguous CLB columns (the stripe) that is able to accommodate all components. The number of columns assigned to a stripe is determined by the width of the components, and by the compatibility of the component resources with the destination area. The stripe may be wider than its widest component, if a component has to be moved to an area that contains the special resources that it uses, like block RAMs or embedded multipliers (fig. 1b). Such components will not be necessarily placed at the left edge of the stripe. Possibly unused spaces in the stripe are filled with feed-through components. These special components simply connect their inputs directly to their outputs. They are also used to provide a path through a stripe when connecting components that are not in adjacent levels. Feed-through components are generated automatically as necessary (no previous partial bitstream is required).

At this stage, an intermediate partial configuration is produced: it is created by merging the partial bitstream of the empty reserved area with the relocated bitstreams of the components (using the approach described in [3]).

The assignment of a component to a location will fail if the total height of the components, including feed-through components added while processing previous levels, is greater than the height of the target area.

The second main stage of the generation procedure is to create the connections among components. Physically,

component terminals are pins of the switch matrix of the corresponding CLB. A connection is defined by the tree-like sequence of switch matrix pins required to establish the desired connectivity from source (an output terminal) to all sinks (input terminals).

The switch matrix pins for each connection are found by a breadth-first search of the area between two adjacent stripes. This routing area is modeled by an array of switch matrices, one for each matrix in the area. For directly abutting stripes, two columns of switch matrices are necessary: one belonging to the right border of the left stripe, and the other belonging to the left border of the right stripe. An extra column of switch matrices is included when there is an unused BRAM column between the stripes.

The actual area searched starts as the smallest rectangle of switch matrices that encloses all terminals of the connection to be routed, and is reduced during the search, limiting the number of segments to be considered. This approach causes some segments not to be considered, but reduces the search effort significantly.

The search starts from the source terminal and is progressively expanded until a destination terminal is reached. Then, a path to the signal source is found by retracing through the sequence of examined interconnection segments. The search is resumed until all sinks of the current connection are reached.

The current implementation adopts the policy of not searching for alternatives when a connecting path cannot be found (for instance, by undoing previous connections and retrying the search), in order to limit the computational effort. In this way, it is possible to create new configurations in a relatively short time. However, the process does not ensure that a global optimum for the complete circuit is obtained, since each one is handled in isolation, without considering the impact on subsequent connections.

After all connections are processed, the partial configuration is updated with the new data for the switch matrices. The resulting partial configuration can then be used to reconfigure the target area of the FPGA.

IV. Experimental Results

For evaluation purposes, the approach described previously was applied to circuits for several arithmetic expressions (on 8-bit data). The structural details of each circuit are summarized in table I. The six circuits tg01-tg06 come from [10]. The corresponding expressions are:

- tg01 $= (a \times b) + (c \times d + e)$
- tg02 $= ((a+b)+(c+d)) \times (((e \times f)+(g+h)) \times (i \times j))$
- tg03 $= ((a \times b) + (c \times d)) \times (e + f)$
- tg04 $= (a+b) \times ((c+d) \times ((e+f) \times ((g \times h) \times (i+j))))$
- tg05 $= ((a+b) \times (c \times d)) \times (((e+f)+(g+h)) \times (i+j))$
- tg06 $= (((a \times b) \times c) + ((d+e) \times f)) +$ $(((g+h)+(i \times j)) + (k \times l \times m))$

Each expression is mapped to a binary tree, whose leaf nodes are the expression variables and whose root node represents the result of the expression. All internal nodes specify binary operations.

TABLE I
Basic structural characteristics of the example circuits

Circuit	# inputs (bits)	# outputs (bits)	# levels	# 8-bit modules	# nets
tg01	40	8	4	9	72
tg02	80	8	5	19	152
tg03	48	8	4	11	88
tg04	80	8	6	19	152
tg05	80	8	5	19	152
tg06	120	8	5	25	200
duo01	48	16	3	7	128
duo02	40	16	4	10	160
duo03	48	16	3	9	144
trio	32	24	5	15	208

Instances tg01 to tg06 are from [10].

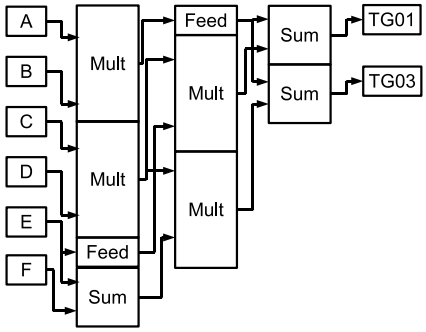

Fig. 2. Component placement for dual01 circuit. (Mult: 8-bit multiplier, Sum: 8-bit adder, Feed: 8-bit feed-through.)

The other benchmark circuits implement more than one expression simultaneously:

- duo01: tg01 and tg03;
- duo02: tg01 and F;
- duo03: tg03 and G;
- trio: F, G and H (with shared common sub-expressions);

where

- $F = a \times b + 3 \times (c + d) - (a \ll 2) + b$;
- $G = 3 \times a \times b + ((b - c) \ll 4)$;
- $H = 5 \times (b - c) + ((c + d) \ll 2) - b$.

The program used to run the benchmarks was written in C and compiled with the GNU Compiler version 3.4.1 included in EDK 8.2. The resulting program has 105 kB of instructions and 1597 kB of static data. As an example, figure 2 shows the floorplan obtained for the "dual01" circuit.

Table II summarizes the results obtained by the proposed procedure for generation of partial configurations, showing the total time required for bitstream generation, the number of levels of the corresponding circuit, the smallest rectangular area occupied by the resulting circuit, the number of feed-through CLBs added during routing, and the number of CLBs taken up by all components (including feed-throughs).

For Virtex-II Pro FPGAs the size of the partial bitstream, and therefore the time taking for partial reconfiguration, is proportional to the number of columns occupied by the circuit.

TABLE II

RESULTS OF THE EXECUTION OF THE PLACEMENT AND ROUTING ALGORITHMS (300 MHZ POWERPC 405, XC2VP30-7 FPGA)

Circuit	Time (s)	Bounding box (Cols×Rows)	# Feed-throughs	Component area (CLBs)
tg01	18.84	10x11	0	71
tg02	57.78	13x17	0	151
tg03	22.03	10x18	0	96
tg04	54.87	16x17	0	166
tg05	55.69	13x17	0	151
tg06	66.94	13x27	0	211
dual01	47.56	10x20	2	84
dual02	59.34	12x23	5	120
dual03	54.48	10x22	1	108
trio	68.75	15x31	10	180

All examples fit in the reserved area of our demonstration system, which is 22 columns by 32 rows.

The example circuits contain from 7 to 21 components (average: 14 components), and from 72 to 208 connections. For this set of circuits, the complete process of bitstream generation takes between 19 s and 69 s (average 48 s) on a PowerPC 405 microprocessor clocked at 300 MHz.

The running time is completely determined by the routing stage. The most time-consuming placement took only 52 ms (for the "trio" circuit). For a circuit with L levels, the procedure for interconnection generation is called $L - 1$ times for the connections between stripes, and two more times to connect primary inputs and outputs to the fixed circuitry.

A one-time reduction in running time can be obtained by using both CPU cores available on the FPGA: since the routing areas between stripes can be processed independently, routing may be performed concurrently by both processors. If partial configurations are reused during the same application run, the use of a configuration cache will avoid repeated generation.

There are several application scenarios that can accommodate delays in the range under discussion. They include applications that must adapt to relatively slow-changing environments (like exterior lighting conditions or temperature) or that may operate temporarily with reduced quality. Another scenario involves adaptive systems that use learning (for instance, of new filter settings) to improve their performance: the time required for generating new configurations may be only a fraction of the time necessary to learn the new settings and to take the decision to switch configurations.

V. CONCLUSION

This paper describes and evaluates a method to generate partial bitstreams at run-time for the dynamic reconfiguration of sections of a Virtex-II Pro platform FPGA. With the goal of obtaining useful solutions in a short time, the proposed approach applies a placement heuristic based on the topological order of arithmetic operators. A router based on breadth-first search over restricted areas determines routes for the interconnections. A partial bitstream implementing arithmetic expressions is constructed by merging together a default bitstream of the reconfigurable area, the relocated partial bitstreams of the components, and the configuration of the switch matrices used for routing. The computational effort is kept in bounds by the use of coarse-grained components, together with a simplified resource model, a direct placement procedure, and the restriction of routing to limited areas.

The proof-of-concept implementation described here shows that run-time generation of configurations is a feasible technique for use in embedded systems, where it can provide tailored hardware support to tasks whose computational needs exceed the capabilities of the CPU. The time required for processing the interconnections makes the current version of this approach unsuitable for applications requiring very fast generation of bitstreams, but several other classes of applications may be able to accommodate the delays involved and profit from the increased flexibility.

Future work includes devising better ways of meeting the conflicting goals of shorter running time and more flexible placement and routing, and of accounting for timing constraints.

ACKNOWLEDGMENTS

The authors would like to thank C. Ababei for providing some of the benchmarks used in section IV.

This work was partially supported by contract PTDC/EEA-ELC/69394/2006 from the Foundation for Science and Technology (FCT), Portugal. Miguel L. Silva was funded by FCT scholarship SFRH/BD/17029/2004.

REFERENCES

[1] M. French, E. Anderson, and D. Kang, "Autonomous system on a chip adaptation through partial runtime reconfiguration," in *16th Intl. Symp. on Field-Programmable Custom Comp. Mach.*, 2008, pp. 77–86.

[2] K. Paulsson, M. Hiibner, J. Becker, J. Philippe, and C. Gamrat, "On-line routing of reconfigurable functions for future self-adaptive systems - investigations within the ÆTHER project," in *Intl. Conf. Field Programmable Logic Appl.*, 2007, pp. 415–422.

[3] M. L. Silva and J. C. Ferreira, "Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems," *IET Computers & Digital Techniques*, vol. 1, no. 5, pp. 461–471, 2007.

[4] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Proc. 39th Design Automation Conference*, 2002, pp. 343–348.

[5] Y. Krasteva, E. de la Torre, T. Riesgo, and D. Joly, "Virtex II FPGA bitstream manipulation: Application to reconfiguration control systems," in *Proc. Intl. Conf. Field Programmable Logic Appl.*, 2006, pp. 1–4.

[6] J. Suris, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAS," in *Proc. Intl. Conf. Field Programmable Logic Appl.*, 2008, pp. 125–130.

[7] M. L. Silva and J. C. Ferreira, "Generation of partial FPGA configurations at run-time," in *Proc. Intl. Conf. Field Programmable Logic Appl.*, 2008, pp. 367–372.

[8] *Virtex-II Platform FPGA User Guide*, Xilinx, Nov. 2007, version 2.2.

[9] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx, Nov. 2007, version 4.7.

[10] C. Ababei and K. Bazargan, "Non-contiguous linear placement for reconfigurable fabrics," *Intl. J. Embedded Systems*, vol. 2, no. 1/2, pp. 86–94, 2006.