2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools

Creation of Partial FPGA Configurations at Run-Time

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*—This paper describes and evaluates a method to generate partial FPGA configurations at run-time. The proposed technique is aimed at adaptive embedded systems that employ run-time reconfiguration to achieve high flexibility and performance. The approach is based on the availability of a library of partial bitstreams for a set of basic components. New partial configurations for circuits defined by netlists of basic components are created by merging together a default bitstream of the target area, the relocated configurations of the components, and the configurations of the switch matrices used for building the connections between the components. An implementation targeting the Virtex-II Pro platform FPGA is described. It runs on the embedded 300 MHz PowerPC CPU present in the FPGA. The proof-of-concept implementation was used to create partial configurations at run-time for 20 circuits with up to 21 components and 288 connections. The complete configuration creation process took between 7 s and 97 s.**

I. Introduction

The growing pervasiveness of computing in all aspects of human life implies the increased importance of autonomous embedded systems that are able to modify their behavior in response to changes in the environment or in the system's goals [1]. Dynamically-reconfigurable hardware is a natural implementation platform for such systems, because it provides the capability to adapt the hardware infrastructure the changing demands. Since embedded systems are resource-constrained (when compared to a regular desktop system), the possibility of reusing the hardware for supporting different tasks at run-time is a very attractive proposition.

Run-time reconfiguration (RTR) of FPGAs is mostly done using partial bitstreams created at design time. A more flexible scheme using run-time creation of bitstreams is justified if creation at design time is impractical or impossible: there may be too many possibilities (e.g., shape-adaptive video processing [2]), or the required information may be only available at run-time (e.g., self-adaptive systems [3]).

This paper describes work on the generation of partial bitstreams at run-time. It assumes the presence in the system of a CPU (in order to run the procedure for creating the new partial bitstreams), and the capability of loading the partial bitstream to a specific FPGA area (without disturbing the operation of other parts of the system). In the prototype described in this paper, we use a Xilinx Virtex-II Pro platform FPGA equipped with a PowerPC processor core. The bitstreams created at run-time are used to modify part of the same FPGA (self-reconfiguration).

The proposed approach starts with a directed acyclic graph (DAG) that describes the connections among medium-sized components (like adders, comparators, and multipliers). The problem of defining the circuit at run-time (selection of the components and definition of their interconnections) is not addressed in this paper.

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., block RAMs). Components can be automatically placed in the target area by the run-time support system, so as to satisfy the constraints imposed by the resource distribution of the reconfigurable fabric. The partial bitstreams of the placed components are merged together (after relocation) with the bitstream of the target area in order to create a single partial bitstream. This is then further modified to include the interconnections among the components, and between the components and the target area's I/O terminals.

The main goal of the implementation is to obtain acceptable solutions in a reasonable time when executing in embedded systems with limited computing resources. Therefore, placement uses a greedy strategy based on the topological order of the components. Connections are established by finding the shortest path from a source terminal to the target terminal for successive nets.

The generation of partial configurations is, by necessity, closely tied to the organization of the underlying reconfigurable fabric, and to the methods available for accessing the configuration memory. Our proof-of-concept implementation runs on a Virtex-II Pro FPGA [4], a device that supports active partial reconfiguration, and has an internal access port for partial device configuration. Other device families from the same vendor (like Virtex-4 and Virtex-5) have similar capabilities, and could, therefore, be targeted in a similar way.

We present results for the generation of partial configurations for a set of 20 circuits with different topologies. The example circuits contain from 3 to 21 components (average: 10 components), and from 32 to 320 connections. For members

978-0-7695-4171-6/10 $26.00 © 2010 IEEE
DOI 10.1109/DSD.2010.14

of this set of circuits, the complete process of bitstream generation takes between 7 s and 100 s (average 47 s) on the embedded PowerPC 405 microprocessor clocked at 300 MHz.

The rest of the paper is organized as follows. Section II describes briefly background and related work. A short overview of the context considered in this work is given in section III. Section IV describes how the reconfigurable infrastructure is modeled for the purposes of bitstream creation. Section V presents the approach to placement and routing implemented in the demonstrator system. Results for the benchmark circuits are detailed in section VI, and concluding remarks are presented in section VII.

II. Related Work

Run-time reconfiguration allows the postponement of some implementation decisions until execution time in order to obtain more flexible systems. Since late binding of functionality to resources generally comes at a cost in performance, one goal of RTR is to offset or avoid this performance loss while preserving flexibility. As is usual in reconfigurable computing, performance improvements are achieved by using logic circuits specialized to the specific computations required at any given time [5]. Flexibility is preserved by the capability to exchange the specialized circuits as needed.

The use of RTR naturally raises the issue of creating the required partial configurations. This is typically done at design time: all necessary partial configurations must be specified and created before the application is deployed [6], [7].

Partial configurations target a specific FPGA area. If another area is targeted after creation, the bitstream must be relocated to address the new target area. This capability makes for more flexible system deployment, so several approaches to the relocation of partial bitstreams have been proposed, including both software tools [8], [9] and hardware solutions [10], [11]. Bitstream relocation is also explicitly included in recent design flows [12].

In most cases, the synthesis tools must be run for each partial configuration. This may be a problem, if many configurations are required, since it is a time-consuming process. A solution based on building a partial bitstreams by combining bitstreams of smaller components is described in [13]. The creation of the new bitstreams requires assigning positions of the target area to components, relocating and merging the individual component bitstreams. In this approach, components connect by abutment, so they must be equipped with matching I/O ports. Since this approach does not rely on the synthesis of logic descriptions, it is a good candidate for implementation in an embedded system for creation of partial configurations at run-time.

A channel router for the Wires-on-Demand RTR framework is described in [14]. It uses a simplified resource database and simple algorithms to find local routes between blocks using relatively few computational resources. Results obtained with a 2.8 GHz Pentium 4 computer indicate that, compared to vendor tools, memory consumption during execution is three orders of magnitude smaller and execution is four orders of magnitude faster, for an average increase in delay of 15% (over a set of seven small benchmarks). The reported implementation results were obtained on a desktop; the possibility of running in an embedded system is mentioned, but no results are presented.

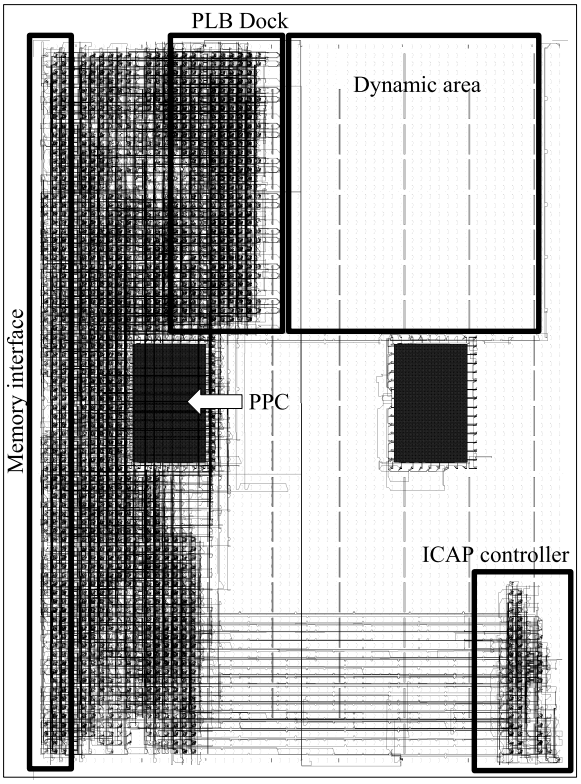

Fig. 1. Floorplan of the hardware platform used for the proof-of-concept implementation. The reserved area for loading partial configurations is shown on the top right (the "dynamic area"). The "PLB dock" implements the interface between the reserved area and the fixed logic. Only the left CPU is used.

A less versatile version of the bitstream assembly approach applied to run-time generation of configurations is described in [15]. In that implementation, inter-module connections are selected from a table of predetermined routes. Although fast, the approach has limited flexibility. The present work does not use of a predetermined set of routes in order to increase flexibility, and includes support for automatic placement of the modules.

III. Application Context

For this work, we assume that the hardware infrastructure has the capability of loading the partial bitstream to a specific FPGA area (without disturbing the operation of other parts of the system). The system should have at least one reserved area for use by the loaded components. This *dynamic area* must be completely unused in the base system. A partial bitstream for this unused area will also be required. In our demonstration system (see fig. 1), a single reserved area is connected to the processor's local bus (PLB), in order to enable fast data transfers between the CPU and the dynamically reconfigured

modules. The block called "PLB dock" implements the interface between the reserved area and the fixed logic.

The partial configurations loaded to a dynamic area are created at run-time by combining the partial bitstreams of smaller modules (the"components"). These are created from RTL descriptions by using standard vendor synthesis tools. Component designs must be restricted to a specific area of the device by specifying the appropriate constraints. Their exact position is not relevant, because the component will be relocated as required for the assembled configuration. We assume that input and output terminals are located on the component's periphery. Each component employs the LUT-based interface macros described in [16]. A component may also use dedicated resources like block RAMs and multiplier blocks.

The bitstream manipulation tool of [13] is used to extract the partial bitstream and to collect additional information. All the information about a component is stored in a file: in addition to the bitstream data, this includes information about the width and height of the component (in CLBs), and about the relative positions of input and output terminals. Component description files are grouped together in component libraries. Reference [13] contains more information about the design process, including a discussion of the issues related to the physical implementation of the terminals.

At run-time, applications can use the code library that we developed in order to assemble new partial configurations using components from all available libraries.

IV. Resource Modeling

The basic element used in the creation of configurations is the rectangular-shaped component with all its terminals on the left or right sides. Components are considered as black boxes during creation of the new configurations: no overlap of components is allowed and no interconnections can traverse them. The use of (medium-sized) components means that applications can view the hardware in terms of elements that are meaningful to the application, instead of being tied to the low-level organization of the logic resources. Another important aspect, is that this approach allows us to limit the amount of information that must be handled at run-time, and therefore reduces the demands imposed on the limited computational resources of a typical embedded system.

The application may specify the location of the modules, or it may use the code library's placement routine. In any case, components should be grouped in vertical stripes. The position of a component inside a stripe and the width of the stripe depend both on the physical resources used by the components and on the position of the stripe in the host area. We also restrict routing to connections between components in adjacent stripes. This restriction simplifies the process of creating the interconnections, because these cannot extend beyond a well-defined free area, a situation that imposes a bound on the corresponding search effort.

All connections are unidirectional: terminals are either inputs or outputs. The output terminals of one component connect to one or more terminals of other components on the next stripe. The terminals to be connected are typically located in adjacent CLB columns. If there are more columns between them, these columns must be empty. In order to limit the effort during routing, only one additional empty column is currently allowed; this is necessary to account for constraints imposed by unused block RAMs and multiplier blocks.

The Virtex-II Pro FPGA has a segmented interconnection architecture: interconnections are built from segments connected through a regular array of switch matrices. These are also connected to the other resources (like CLBs and BRAMs) [17]. From the large number of routing resources available in the reconfigurable fabric, we use the following subset:

- 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 or width) are not used since they can interfere with circuitry outside of the host area. Horizontal hex lines reach beyond the area allowed for the connections, which has only up to three columns of switch matrices.

The final model of the switch matrix contains 116 pins, distributed as follows:

- 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.

The run-time support library models the area used for the connections as a two-dimensional array of switch matrices, and employs a data structure based on the model just described to keep track of switch matrix resource usage.

V. Partial Bitstream Creation

The run-time creation of new partial configuration starts from a component netlist, which specifies the components to be used and the (unidirectional) connections between their terminals. No cycles between the components are allowed, i.e., the data flow must be represented by a directed acyclic graph. The creation proceeds in two stages: 1) definition of component locations; 2) creation of connections (including the connections to the interface of the host area).

A. Selecting Component Locations

The current strategy for determining the location of a component groups components in columns (stripes), so that directly connected components are assigned to adjacent groups if possible. The arrangement in columns matches the reconfiguration mechanism of Virtex-II-Pro FPGAs, where the

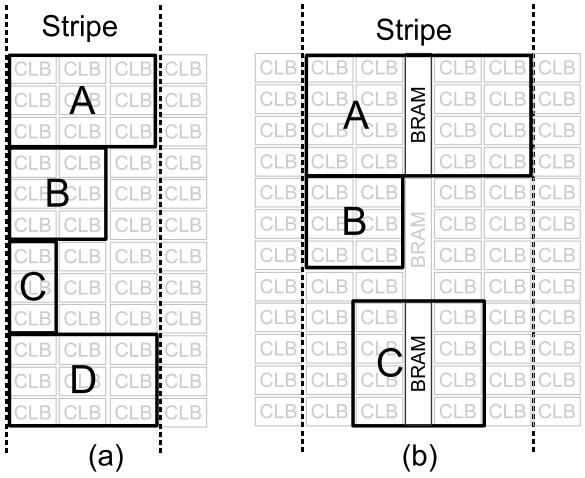

Fig. 2. Placing components in stripes. (a) Typical placement for components that only have CLBs; (b) Placement resulting from restrictions imposed by the use of particular hardware resources (block RAMs in this case).

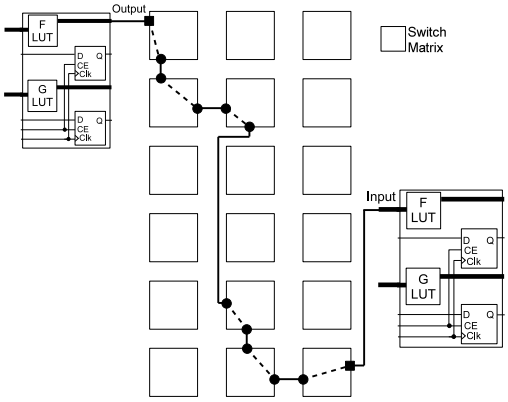

Fig. 3. Example of a route interconnecting two components. (Drawing elements are not to scale.)

smallest unit of reconfiguration data applies to an entire column of resources. Two examples of possible arrangements of components in a stripe are displayed in figure 2.

The first step in grouping is to determine each component's level (counted from the primary inputs). The first level contains the components whose inputs are connected to the PLB dock; the second level contains those 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 the highest-numbered component connected to it. This procedure assigns component levels in topological order.

The next step is to determine the set of contiguous CLB columns (a stripe) required for all components of each level. The final placement of a component will be restricted to the columns assigned to its level. Levels are processed in order. The starting column assigned to a given stripe will be the one closest to the PLB dock without overlapping previous stripes. The number of columns assigned to a stripe is the smallest one required to accommodate all components of the corresponding level (see Figure 2a). This is determined by the width of the components and by the compatibility of the component resources with the destination area. In some cases it is necessary to widen the stripe in order to cover an area compatible with the resource requirements of a given component (see fig. 2b).

The detailed assignment of components to locations is done by processing each level in succession, and placing the components from top to bottom in the device. The placement of components with non-homogeneous resources (like BRAMs) may require offsetting the component from the default location. As a result, the stripe may have unused areas at its left or right borders (fig. 2b).

The unused areas of the stripe are filled with feed-through components, in order to ensure that all inputs are available at the left border of the stripe, and that all outputs are brought to the border on the right. Feed-through components simply connect their inputs directly to their outputs. Components of this type are also used to provide a path through a stripe when connecting components that do not belong to the same level. Feed-through components are generated as required, without recourse to library components.

The assignment of a component to a location fails if the sum of the heights of all components of the same level, including feed-through components added while processing previous levels, is greater than the height of the host area.

This stage produces a partial configuration which is obtained by merging the partial bitstream of the empty host area with the relocated bitstreams of the components.

B. Connecting the Components

The second stage of the run-time creation of a new partial configuration determines which interconnect resources are assigned to each connection between components. Given the previously-described strategy for defining the locations of the components, this can done by finding out how to establish connections between terminals of components in adjacent stripes.

Since we are using LUT-based bus macros for implementing the terminals as described in [16], component terminals correspond to pins of a switch matrix. The pins of a switch matrix are connected to pins of other switch matrices according to the resource model of section IV.

One connection between two components is defined by the sequence of switch matrix pins required to establish the desired connectivity. For each matrix, this sequence defines the internal connections that are required, and therefore defines the configuration settings of the switch matrices involved. Figure 3 illustrates one such connection. The dots represent the internal pins, while the dashed lines depict the internal connections that must be established in each switch matrix.

In order to determine all the pins involved in a connection, a breadth-first search of the routing area is performed. The routing area is represented by an array of switch matrices. For

adjacent 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/multiplier column between the stripes.

The actual area searched starts as the smallest rectangle that encloses all pins used as terminals of the connection to be established, and is reduced during the search, thereby limiting the number of segments to are considered. Restricting the search area in this way may cause some segments to be left out of consideration, but reduces the search effort significantly. Connections are processed in sequence and no retrying of failed searches is performed.

The shortest path from a source (output terminal) to the corresponding sinks (one or more input terminals) is found by a variant of Dijkstra's shortest path algorithm [18]. The breadth-first search procedure maintains a list of those pins that can be reached from the source by using exactly a number of segments equal to current iteration count. For any pin on this list, there is a shortest path (measured in number of segments) to the source. The search is managed so that a pin can enter this list only once (at the earliest opportunity).

When a sink is reached, a path to the source is determined by retracing through the sequence of interconnection segments. The search is resumed until all sinks of the current connection are reached.

All connections are processed sequentially in this way. Pins used for a connection cannot be used in subsequent searches. Therefore, the order in which connections are processed may influence the final result. (Evaluation of the influence of this factor is outside the scope of the present paper.)

After all connections are processed, the partial configuration is updated with the configuration information for the new routes.

This algorithm does not ensure that a global optimum for all routes is obtained, since each net is handled individually, without considering the influence on the routing of the following nets. The impact of these limitations is reduced by the fact that choices at this stage are considerably restricted by the previous placement, and by the design decision to keep any interconnections confined to the area between stripes. As the next section shows, several classes of circuits can be successfully routed under these restrictions.

VI. Experimental results

The algorithms of the previous section were applied to 20 synthetic circuits. The evaluation was done with a XUP Virtex-II Pro Development System, which has a Xilinx XC2VP30-7 FPGA [4] and 512 MB of external DDR memory (PC-3200). The external memory contains the program code and data, including the library of components. Only one of the two embedded PowerPC 405 processor cores is used (running at 300 MHz). The 64-bit processor local bus connected to the memory controller uses a 100 MHz bus clock. 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.

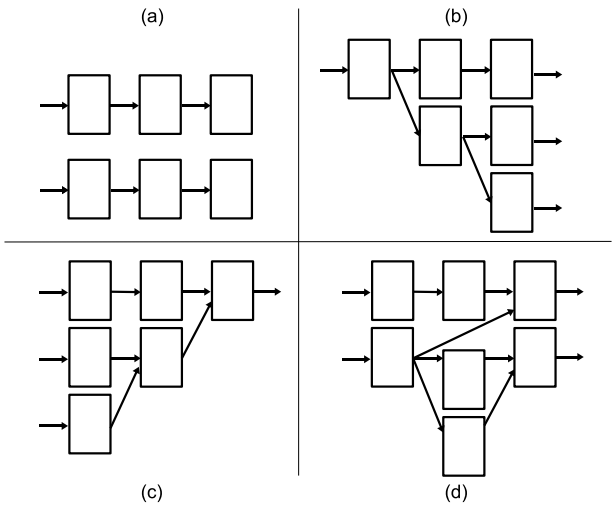

Fig. 4. Structure of connection graphs for each circuit class used in the evaluation of the demonstrator. (a) One or more pipelines; (b) Tree-like graphs with a single input component and multiple output components; (c) Tree-like graphs with multiple input components and a single output component; (d) Random directed acyclic graphs.

The resulting programs has 105 kB of instructions and 1597 kB of static data.

The set of test circuits comprises four classes, whose general structure is depicted in fig. 4:

1) **Pipeline:** The netlists represent one or more pipelines. Pipeline components have 8-bit or 16-bit ports, each pipeline has three or four levels, and each example contains between one and three pipelines operating in parallel.
2) **Tree SM:** The netlists represent trees with one input component and multiple output components. Components have 8-bit, 16-bit or 32-bit ports. Examples have tree structures with two to four levels, and components connect to between one and four components on the next level.
3) **Tree MS:** The elements of this class of circuits are trees with multiple input components and a single output component. Again, components have 8-bit, 16-bit or 32-bit ports. The set includes examples with two to four levels of components. These circuits display a structure similar to arithmetic expressions (with binary operators).
4) **Random DAG:** These are netlists for random directed acyclic graphs using components with 8-bit, 16-bit and 32-bit ports. The number of components used in the examples ranges from 5 to 13.

The structure of the first three classes is well matched to the placement strategy, while the last class is more general. Table I summarizes the structural characteristics of the circuits: number of input and output ports, number of components for each of the three different data sizes (8, 16 and 32 bits), number of levels of the structure, and maximum fan-out (number of sinks of a net).

TABLE I
BASIC STRUCTURAL CHARACTERISTICS OF THE CIRCUITS USED FOR EVALUATION

Circuit	Number of Inputs	Number of Outputs	Levels	Number of Modules			Nets	Maximum fan-out
				8-bit	16-bit	32-bit		
Pipeline 1	8	8	3	3			32	1
Pipeline 2	16	16	3	6			64	1
Pipeline 3	24	24	4	12			120	1
Pipeline 4	24	24	4	4	4		120	1
Pipeline 5	32	32	4	4	4	1	160	1
Tree sm1	8	16	2	3			40	2
Tree sm2	8	32	3	7			88	2
Tree sm3	16	64	3	4	5		176	2
Tree sm4	32	64	4	12	2	3	288	2
Tree sm5	32	64	4	16	4	1	288	4
Tree ms1	32	8	2		3		48	1
Tree ms2	32	16	3	3			128	1
Tree ms3	64	32	3	4	5	1	224	1
Tree ms4	64	32	4	12	2	1	288	1
Tree ms5	64	8	4	12	8	1	320	1
Random DAG 1	16	8	3	5			72	2
Random DAG 2	32	32	3	5	2		112	2
Random DAG 3	32	32	4	7	5		208	2
Random DAG 4	32	32	5	5	6	1	264	2
Random DAG 5	32	32	5	7	4	1	256	4

TABLE II
EXECUTION TIME FOR CONFIGURATION GENERATION ON THE 300 MHZ POWERPC 405 EMBEDDED IN THE VIRTEX-II PRO XC2VP30-7 FPGA

Circuit	Time (s)	Bounding box (Columns x Rows)	Number of feed-throughs	Total component area (CLBs)	Area used for feed-throughs (%)
Pipeline 1	6.97	6x3	0	18	0
Pipeline 2	13.67	6x6	0	36	0
Pipeline 3	23.94	8x12	0	96	0
Pipeline 4	26.63	12x9	4	96	4
Pipeline 5	39.11	12x12	4	132	3
Tree sm1	9.39	6x6	1	24	4
Tree sm2	20.92	8x12	4	66	6
Tree sm3	45.23	9x24	4	132	3
Tree sm4	92.85	11x24	4	180	2
Tree sm5	99.73	12x24	12	192	6
Tree ms1	10.73	5x8	0	30	0
Tree ms2	27.14	9x12	2	90	2
Tree ms3	73.54	9x24	4	144	3
Tree ms4	94.17	12x24	8	216	4
Tree ms5	96.90	12x32	16	256	6
Random DAG 1	16.28	8x6	2	46	4
Random DAG 2	26.21	9x16	6	102	6
Random DAG 3	50.35	12x24	12	156	7
Random DAG 4	86.44	16x32	22	185	11
Random DAG 5	90.21	21x32	30	164	15

Table II summarizes the results obtained with the proposed approach to configuration generation at run-time: total time required for bitstream generation, the smallest rectangular area occupied by the resulting circuit, the number of feed-through CLBs added during routing, the number of CLBs taken up by all components (including feed-throughs), and the percentage of area occupied by feed-through components.

The total running time is completely determined by the routing stage: the most time-consuming placement took 114 ms (for the "Random DAG 5" circuit). The procedure for creation of connections is called $L+1$ times for each circuit, where L is the number of levels: $L-1$ times for the connections between stripes, and two more times for connecting the primary inputs and outputs to the PLB dock. For Virtex-II Pro FPGAs the size of the partial bitstream, and therefore the time taken by partial reconfiguration, is proportional to the number of columns occupied by the circuit (first number in the third column). For the platform used, each column takes 0.31 ms to reconfigure. All circuits fit in the host area of our test system, which is 22 columns by 32 rows.

The placement of components may involve adding feed-through components to the circuit in order to connect components that are not on successive levels (cf. section V-A). With the exception of two benchmarks (the two largest random DAGs), the additional components represent less than 10% of the total number of CLBs used by all components.

Most benchmarks took less than 90 s; the exceptions are the two of the largest trees (of both types), and the largest random DAG. The global average running time is 47.5 s. The most time consuming example took 99.73 s ("tree sm5").

For the hardware setup used in this evaluation, a one-time reduction in running time can be obtained by using both CPU cores: since the areas between stripes can be processed independently, connection generation may be easily performed concurrently by both processors.

Another possibility to improve global system performance, applicable when partial configurations are reused during the same application run, is to maintain a configuration cache.

The running times achieved by the current implementation make it unsuitable for applications that require very fast creation of configurations, like just-in-time hardware compilation. However, there are many application scenarios that may 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 configurations may be only a part of the time necessary to learn the new settings and to take the decision to switch configurations.

VII. Conclusion

This paper describes and evaluates a method to generate partial bitstreams at run-time for use with dynamically reconfigurable FPGAs. The main goal is to obtain useful solutions in a short time. The computational effort is limited by several design choices: circuit description by directed acyclic graphs of coarse-grained components, simplified resource models, direct placement procedure, and use of limited areas for routing.

The results for a set of circuits show that the time required for bitstream generation on a 300 MHz PowerPC embedded processor depends strongly on the complexity of the circuits, averaging 47.5 s (minimum: 6.97 s, maximum: 99.73 s) for an average circuit size of 10 components (minimum: 3, maximum: 21) and 164 connections (minimum: 32, maximum: 320).

The working implementation described here shows that run-time generation of configurations is a feasible technique that can be used in embedded systems to provide hardware for very specialized tasks.

The evaluation of the suitability of this approach for specific applications requires that all system aspects be considered. For the current implementation, the time required restricts its application to systems that can gracefully handle the delays involved (for instance, because temporarily degraded performance is acceptable), or to situations where the need for new configurations may be predicted (at run-time) with some advance. It may also be possible to amortize the generation time by caching configurations for reuse.

The global performance achieved by the implementation, although encouraging, indicates that further work is necessary to meet the conflicting goals of shorter running time and more flexible placement and routing. In addition, the approach will be expanded to take timing information and constraints into account.

Acknowledgments

The present work was partially supported by research 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 International Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, 2008, pp. 77–86.

[2] J. Gause, P. Cheung, and W. Luk, "Reconfigurable computing for shape-adaptive video processing," *IEE Proceedings - Computers and Digital Techniques*, vol. 151, no. 5, pp. 313–320, 2004.

[3] 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 *International Conference on Field Programmable Logic and Applications (FPL 2007)*, 2007, pp. 415–422.

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

[5] S. Hauck, "The roles of FPGAs in reprogrammable systems," *Proc. IEEE*, vol. 86, no. 4, pp. 615 –638, Apr. 1998.

[6] I. Robertson and J. Irvine, "A design flow for partially reconfigurable hardware," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 257–283, 2004.

[7] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2006)*, 2006, pp. 1–6.

[8] 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.

[9] Y. Krasteva, E. de la Torre, T. Riesgo, and D. Joly, "Virtex II FPGA bitstream manipulation: Application to reconfiguration control systems," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2006)*, 2006, pp. 1–4.

[10] H. Kalte and M. Porrmann, "REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs," in *Proceedings of the 3rd Conference on Computing Frontiers*. ACM, 2006, pp. 403–412.

[11] F. Ferrandi, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," in *Proc. International Symposium on System-on-Chip (Soc 2006)*, 2006, pp. 1–4.

[12] H. Tan and R. F. DeMara, "A multilayer framework supporting autonomous run-time partial reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 504–516, 2008.

[13] 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.

[14] J. Suris, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAS," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008, pp. 125–130.

[15] M. L. Silva and J. C. Ferreira, "Generation of partial FPGA configurations at run-time," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008, pp. 367–372.

[16] M. Hübner, T. Becker, and J. Becker, "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration," in *Proc. 17th Symposium on Integrated Circuits and Systems Design (SBCCI 2004)*, Sept. 2004, pp. 28–32.

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

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. McGraw-Hill, Dec. 2003.