Run-time Generation of Partial FPGA Configurations for Subword Operations

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*—Instructions for concurrent processing of smaller data units than whole CPU words are useful in areas like multimedia processing and cryptography. Since the processors used in FPGA-based embedded systems lack support for such applications, this paper proposes mapping sequences of subword operations to a set of hardware components and generating the corresponding FPGA partial configurations at run-time. The technique is aimed at adaptive embedded systems that employ run-time reconfiguration to achieve high flexibility and performance. New partial configurations for circuits implementing sets of subword operations are created by merging together the relocated partial configurations of the hardware components (from a predefined library), and the configurations of the switch matrices used for the connections between the components. The paper presents and discusses results obtained for a 300 MHz PowerPC CPU in a Virtex-II Pro platform FPGA. For the set of benchmarks analyzed, the complete configuration creation process takes between 5 s and 60 s. The run-time generated hardware versions achieved speed-ups between 17 and 73 over the software versions.**

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. Dynamically-reconfigurable hardware is a natural implementation platform for such systems, because it provides the capability to adapt the hardware infrastructure to 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 [1]) or the required information may be only available at run-time (e.g., self-adaptive systems [2]).

Multimedia and cryptography applications are often supported by special-purpose CPU instructions [3], which enable concurrent SIMD (single-instruction multiple-data) processing of smaller data units than whole CPU words (e.g., MMX and SSE instructions for Intel processors, or AltiVec for the PowerPC architecture [4]). Since the processors used in FPGA-based embedded systems usually lack such SIMD extensions, this paper proposes mapping sequences of subword operations to a set of hardware components and generating the corresponding FPGA partial configurations at run-time. It assumes the presence on 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). For the evaluation presented in this paper, we use a Xilinx Virtex-II Pro platform FPGA equipped with a 405 PowerPC processor core (without floating-point or SIMD instructions). The bitstream created at run-time are used to modify part of the same FPGA (self-reconfiguration).

The creation of partial configurations starts with a directed acyclic graph (DAG) that describes the connections among medium-sized components (like adders, comparators, and multipliers). The components can be automatically placed in the target area so that the constraints set by the resource distribution of the reconfigurable fabric are respected. In this work, the DAGs represent the mapping of sets of SIMD instructions to components. 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).

The main goal of the implementation is to obtain acceptable solutions in a reasonable time when executing in embedded systems with limited computing resources. 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 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 [5], a device that supports active partial reconfiguration, and has an internal access port for partial device configuration. We also measured the speed-ups obtained by the generated hardware over the corresponding

software version running on the PowerPC 405.

The paper is organized as follows. Sec. II describes briefly background and related work. A short overview of the application context considered in this work is given in sec. III. Sec. IV describes how the reconfigurable infrastructure is modeled for the purposes of bitstream creation. Sec. V presents the approach to placement and routing implemented in the demonstrator system. Results for several application fragments are detailed in sec. VI, while some final remarks are presented in sec. VII.

II. Related Work

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].

Configurations target a specific FPGA area. If that area changes after creation, partial bitstreams must be relocated to the 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 [7], [8] and hardware solutions [9].

In all 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 by [10]. The creation of the new bitstreams requires assigning positions of the target area to components, relocating and merging the individual component bitstreams, and interconnecting the components by modification of the merged bitstream. 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 [11]. It uses a simplified resource database and simple algorithms to find local routes between blocks using relatively few computational resources: memory consumption during execution is 3 orders of magnitude smaller and execution is 4 orders of magnitude faster than vendor tools (over a set of seven small benchmarks). The reported implementation results were obtained on a desktop PC; the possibility of running in an embedded system is mentioned, but no results are reported.

A less versatile version of the bitstream assembly approach applied to run-time generation of configurations is described in [12]. 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, and includes support for automatic placement of the components.

III. System Architecture

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).

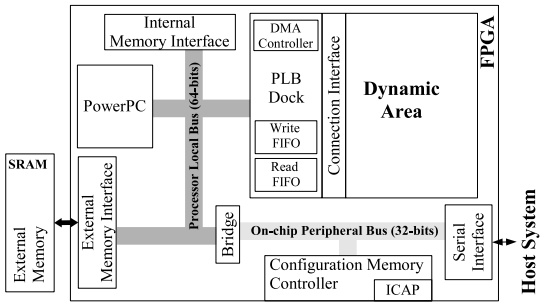

Fig. 1. Architecture of the hardware platform for the prototype implementation. The configuration generation procedure runs on the PowerPC CPU. The reserved area for loading partial configurations is shown on the right (the "dynamic area"). The "PLB dock" implements the interface between the reserved area and the fixed logic.

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 [13]. A component may used dedicated resources like block RAMs and multiplier blocks, but these impose restrictions on relocation.

The bitstream manipulation tool of [10] is used to extract the partial bitstream. 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. More information about the design process can be found in [10]. 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 black boxes during creation of the new configurations: no overlap of components is allowed and no interconnections can traverse

them. Each component implements the basic operation of a SIMD instruction. Instructions that permute or exchange bits only affect the routing of data, and do not require supporting components. The use of medium-sized components limits the amount of information that must be handled at run-time, reducing the load imposed on the limited computational resources available in 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 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 by ensuring that they do not extend beyond a well-defined free area, and by reducing 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 the embedded block RAMs and multipliers.

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) [14]. From the large number of routing resources available in the reconfigurable fabric, we use direct connections to neighbor CLBs, double lines and vertical hex lines. 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 target 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;
- 20 vertical hex lines;
- 8 connections to the outputs of the 4 slices in the CLB;
- 32 connections to the inputs of the 4 slices in the CLB.

The area used for connections is modeled as a two-dimensional array of switch matrices, and employs a data structure based on the simplified model just described to keep track of resource usage.

V. Partial Bitstream Generation

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 netlist must define a directed acyclic graph. The creation proceeds in two stages: 1) defining the component locations; 2) creating the connections (including the connections to the interface of the host area).

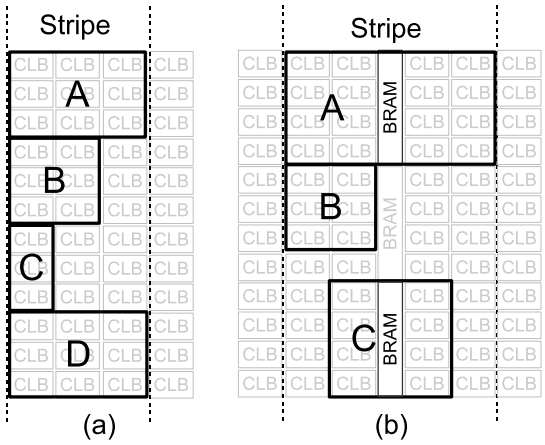

Fig. 2. Placing components in stripes. (a) Typical placement for components that only have CLBs; (b) Placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

A. Determining Component Locations

The current strategy for determining the location of a component groups the 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 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 the components is to determine its level (counted from the primary inputs). The first level contains the components whose inputs are connected to the PLB dock; 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 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 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).

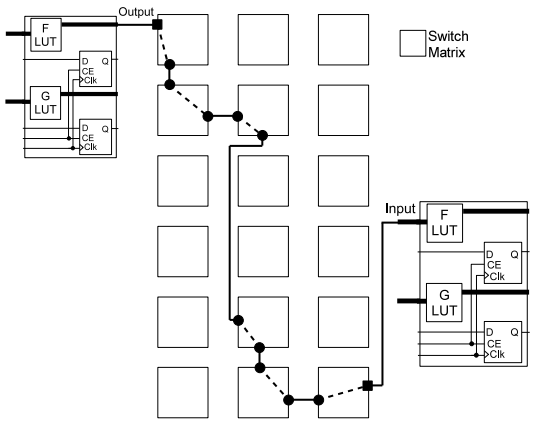

Fig. 3. Example of one possible routed connection.

The detailed assignment of components to locations processes each level in succession, placing the components from top to bottom of 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 right border. 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. Component Interconnections

The second stage of the run-time creation of a new partial configuration must determine which interconnect resources are assigned to each connection between components. Given our strategy for defining the locations of the components, this can done by finding 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 [13], component terminals correspond to some pins of a switch matrix. Other pins in the switch matrix are connected to pins in other switch matrices according to the resource model of sec. 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. Fig. 3 illustrates one such connection. The dots represent the internal pins, while the dashed lines depict the internal connections that must be established.

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 performed by a variant of Dijkstra's shortest path algorithm [15]. The breadth-first search of the area considered for routing 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 information for the new routes.

The search procedure does not ensure that a global optimum for all routes is obtained, since each net is handled in isolation, without considering the impact on the following nets. The impact of these limitations is mitigated by the fact that choices are 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, many sequences of SIMD instructions can be routed under these restrictions.

VI. Evaluation

The algorithms of the previous section were applied to 10 DFG specifications derived from SIMD code fragments. The evaluation was done on a XUP Virtex-II Pro Development System, which has a Xilinx XC2VP30-7 FPGA [5] 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. The resulting programs has 105 kB of instructions and 1597 kB of static data.

The following tasks, which can be efficiently written with SIMD instructions, were selected for evaluation of the proposed approach. The first six process gray-scale images with 8-bit pixel values.

1. Brightness adjustment (BA): The hardware adds a pixel value to a signed constant value (saturating add).

2. Contrast adjustment (CA): The hardware processes each pixel according to $a + (|pixel - a| * c)$, where a and c are constants that represent average brightness and contrast change respectively.

3. Additive blending (AB): This task consists of adding (with saturation) the pixel values from two images to produce a third.

4. Fade effect (FE): This task consists of combining the pixels of two images according to $(A-B)\times f+B$, where A is a pixel value from the first image, B is a pixel value from the second, and f is a constant that specifies the relative contribution of the first image to the result [3]. The fade-in-fade-out effect is obtained by processing the source images successively for different values of f. The structure of the circuit generated for this task is shown in fig. 4.

5. Motion detection (MD): This task combines the pixels of two images to produce a third. The absolute difference between corresponding pixels of the source images is calculated. If the calculated value is above a predefined threshold, the resulting pixel is white; otherwise it is black.

6. Motion overlay (MO): This task again combines the pixels of two images to produce a third. As before, the absolute difference between corresponding pixels is calculated. If the calculated value is above a predefined threshold, the result is equal to the mean of the input pixels; otherwise it is equal to the pixel value from the first image.

7. Clip test (CT): This task is used in clipping off triangles for graphics processing. A comparison between 4 spatial coordinates is performed for 3 vertices. The task is the same for each vertex, so vertices are processed one at a time [16].

8. Least Significant Bit Steganography (LSBS): This task takes a sequence of bytes representing a secret message and encodes them onto 16-bit PCM audio samples by replacing the four least significant bits of each sample with bits from the message (hilewitz08-bit).

9. Compression (CO): This task comes from bio-informatics: compress a vector of string representation of codes of the four nucleotides into vector with a 2-bit representation for each nucleotides [17].

10. Translation (TR): This task translates a compressed sequence of four sets of three nucleotides (6 bits each, as produced by the previous task) to four protein codons, each one represented by an 8-bit.

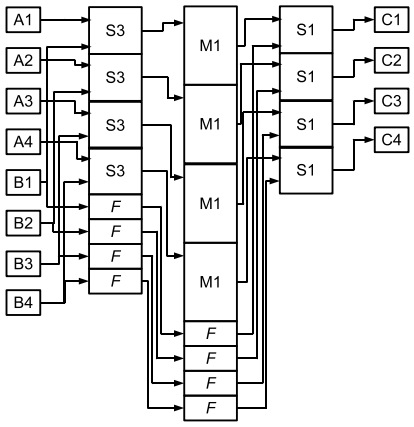

Fig. 4. Component placement for Fade effect circuit. (M1: 8-bit multiplier, S1: 8-bit adder, S3: 8-bit subtractor, Feed: 8-bit feed-through.)

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

Name	In	Out	Lv	Comp.	Nets	Bbox (c×r)	Time (s)
BA	32	32	1	4	64	3x12	15
CA	32	32	3	12	128	9x32	32
AB	64	32	1	4	96	3x12	25
FE	64	32	3	12	128	9x32	38
MD	64	32	2	8	128	6x12	32
MO	64	32	3	16	224	12x28	60
CT	32	6	2	7	176	6x15	46
LSBS	40	32	2	6	128	6x14	36
CO	32	8	1	0	8	3x8	5
TR	24	32	1	0	24	3x8	8

In: number of input bits; *Out:* number of output bits; *Lv:* number of levels; *Comp:* number of components; *Nets:* number of nets routed; *Bbox:* bounding box (CLB columns by CLB rows); *Time:* time for whole generation process.

The last two benchmarks involve only bit selection and permutation. Therefore, they are implemented only by routing, without any logic blocks.

Tab. I summarizes the structural characteristics of the circuits and the total configuration generation time (including placement, routing, and creation of partial bitstream in the standard format).

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.

All benchmarks required less than 60 s total generation time. This is almost completely determined by the routing stage: he most time-consuming placement took less than 100 ms.

Tables II and III compare the running times of software and run-time-generated hardware versions of the benchmarks. The hardware execution time are, in these cases, determined only by the data transfer times (DMA setup time included). In tab. II

TABLE II
EXECUTION TIME PER OUTPUT PIXEL FOR IMAGE PROCESSING TASKS

Tasks	Software (µs)	Hardware (µs)	Speedup
BA	0,48	0,01	48,0
CA	0,73	0,01	73,0
AB	0,64	0,04	16,0
FE	0,90	0,04	22,5
MD	0,79	0,04	19,8
MO	0,82	0,04	20,5

Note: All images are 8-bit, gray-scale, 1024×1024.

TABLE III
EXECUTION TIME FOR NON-IMAGE-PROCESSING TASKS

Name	Software (ms)	Hardware (ms)	Speedup
CT	875,56	26	33,4
LSBS	1207,07	26	46,1
CO	290,29	26	11,1
TR	465,69	26	17,8

Note: Each task executes 250 000 times.

they are different for tasks that process one image (BA, CA) or two images. All tasks in tab. III involve the same amount of data transfers and therefore have the same execution time.

The speed-ups achieved by the hardware are significant, showing that the run-time generation of dedicated hardware may provide significant advantages, provided that the generation times are acceptable for the specific application.

The generation times achieved by this implementation are unsuitable for applications that require very fast turnaround times, 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). that may operate temporarily with reduced quality, or that may amortize the generation effort over a large enough utilization time. If partial configurations are reused during the same application, overall performance may be improved by keeping a cache of previously-generated configurations.

VII. CONCLUSION

This paper evaluated the run-time creation of partial configurations that implement small sets of SIMD instructions for use in embedded systems with dynamically reconfigurable FPGAs. The main goal of the generation procedure is to obtain useful solutions in a short time. The computational effort is bounded by several design choices: circuit description by acyclic netlists of coarse-grained components, simplified resource models, direct placement procedure, and use of limited areas for routing.

For a set of ten SIMD-code-derived benchmarks, the generation times varied between 5 s and 60 s; the generated hardware exhibited speed-ups between 17 and 73 compared to the software versions.

The working implementation described here shows that run-time generation of configurations is a feasible technique for use in embedded systems to provide specialized hardware support to tasks whose computational needs exceed the computational power of the CPU.

Further work is necessary to meet the conflicting goals of shorter running time and more flexible placement and routing. The inclusion of timing information and constraints is also a future goal.

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] 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.

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

[3] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Commun. ACM*, vol. 40, no. 1, pp. 24–38, 1997.

[4] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85–95, 2000.

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

[6] 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. Intl. Conference on Field Programmable Logic and Applications (FPL 2006)*, 2006, pp. 1–6.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] R. B. Lee and A. M. Fiskiran, "PLX: an instruction set architecture and testbed for multimedia information processing," *The Journal of VLSI Signal Processing*, vol. 40, no. 1, pp. 85–108, May 2005.

[17] Y. Hilewitz and R. Lee, "Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors," *Journal of Signal Processing Systems*, vol. 53, no. 1, pp. 145–169, Nov. 2008.