Microprocessors and Microsystems 36 (2012) 365–374

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Run-time generation of partial FPGA configurations for subword operations

Miguel L. Silva [a,*,1], João Canas Ferreira [b]

[a] Faculdade de Engenharia, Universidade do Porto, R. Dr. Roberto Frias, 4200-465 Porto, Portugal
[b] INESC TEC, Faculdade de Engenharia, Universidade do Porto, R. Dr. Roberto Frias, 4200-465 Porto, Portugal

ARTICLE INFO

Article history:
Available online 17 February 2012

Keywords:
Run-time reconfiguration
SIMD instructions
Configuration bitstream
FPGA
Run-time configuration generation
Adaptive embedded systems

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 1 s and 24 s. The run-time generated hardware versions achieve speed-ups between 11 and 73 over the software versions.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The increasing 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 very attractive.

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. For instance, in shape-adaptive video processing [1] there are too many possible configurations to be produced in advance. In other applications, like the self-adaptive system described in [2], the necessary information is only available at run-time.

On the other hand, multimedia and cryptography applications are often supported by special-purpose CPU instructions, which enable concurrent single-instruction multiple-data (SIMD) processing of smaller data units than whole CPU words. Examples are the MMX [3] and SSE [4] instructions of Intel processors, or the AltiVec instructions for the PowerPC architecture [5].

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 at run-time the corresponding FPGA partial configurations at run-time. By focusing on subword operations with integer operands, this approach leverages the parallelism exposed by the software development effort for other platforms (e.g., personal computers). The resulting circuits tend to be very regular and have a straightforward data flow, so that simple place and route algorithms can be used to produce implementations that achieve good speed-ups. In this way, the complexity associated with the creation of more general cores at run-time is avoided.

The proposed approach assumes that the system is capable of loading the partial bitstream to a specific FPGA area without disturbing the operation of other parts of the system (as supported by devices like Virtex-II Pro, Virtex-4 and Virtex-5 from Xilinx [6,7], or Atmel's FPSLIC and AT40K [8]). For the experimental 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 bitstreams created at run-time are used to modify part of the same FPGA (self-reconfiguration). This approach allows the postponement of some implementation decisions until execution time with the objective of obtaining more flexible systems: performance improvements are achieved by having dedicated support for the specific computations required at any given time, while flexibility is preserved by being able to change the specialized circuits as needed.

* Corresponding author.
E-mail addresses: mlms@fe.up.pt (M.L. Silva), jcf@fe.up.pt (J.C. Ferreira).

[1] Funded by Grant SFRH/BD/17029/2004 from Foundation for Science and Technology (FCT), Portugal.

0141-9331/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2012.02.010

The creation of partial configurations starts with a directed acyclic graph (DAG) that describes the connections among coarse-grained components like adders, comparators, and multipliers. The DAGs map sets of SIMD instructions to components, and specify the connections between component terminals. The data flow is assumed to be unidirectional: feedback connections among components are not supported. Each component implements a basic operation supported by typical SIMD instructions. The parallel execution of the operations is achieved by instantiating a component multiple times. 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). Optionally, estimates of the worst-case delay between inputs and outputs may also be included (for combinational components).

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 system automatically places components in the target area so that the constraints set by the resource distribution of the reconfigurable fabric are met. 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 [6], 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 [7]) have similar capabilities, and can, in principle, be targeted in a similar way.

We use a set of benchmarks, mostly from image processing applications, to empirically evaluate the time required to create the partial bitstreams. We also measure the speed-ups obtained by the generated hardware over the corresponding software version running on the PowerPC 405.

The paper is organized as follows. Section 2 presents some related work. A short overview of the application context considered in this work is given in Section 3. Section 4 describes how the reconfigurable infrastructure is modeled for the purposes of routing interconnections between components. Section 5 presents the approach to placement and routing implemented in the demonstrator system. Results for several benchmarks are detailed in Section 6, while final remarks are presented in Section 7.

2. 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 [9]. 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 [10,11] and hardware solutions [12].

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 Silva and Ferreira [13]. 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 use at run-time. Issues associated with keeping the reconfiguration process from affecting other sections of the FPGA are also discussed in that paper.

Just-in-time software compilation has a long history [14]. The main concept is to spend computational resources at run-time in order to generate efficient, specialized versions of subroutines, possibly exploiting additional knowledge that was not available at compile-time. The generation effort is then amortized over the remaining time of application execution. A frequent application of this approach is to accelerate code written for virtual machines [15]. Typically, the generation effort is applied to improve much-used code kernels with large potential impact on performance, and is subject to challenging time constraints.

The concept of just-in-time hardware compilation is introduced in [16]. The objective is to produce hardware for specialized instructions. The initial implementation concept for execution on the Virtex-II Pro FPGA targets a very restricted layout built out of fixed-width 16-bit operators, and uses a fast router based on look-up tables. Place and route times under 100 ms for unspecified expressions are reported. It is unclear how the approach would behave in more realistic scenarios.

The Wires-on-Demand RTR framework [17] includes a channel router, which uses a simplified resource database and simple algorithms to find local routes between blocks while requiring relatively few computational resources: memory consumption during execution is three orders of magnitude smaller and execution is four orders of magnitude faster than when using vendor tools (over a set of seven small benchmarks). The reported implementation results were obtained on a Pentium 4 PC (2.8 MHz); the possibility of running in an embedded system is mentioned, but no results are reported.

The version of the bitstream assembly approach applied to run-time generation of configurations described in [18] is less versatile than the one proposed in this work. 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.

3. System architecture overview

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 defining the default state of this area also 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 PLB dock contains a direct memory access (DMA) controller and two FIFOs. Each FIFO has capacity for 1024 32-bit words. A typical use scenario involves three steps: (1) filling the input FIFO; (2) feeding the circuits in the dynamic area with the contents of the input FIFO together with data from the PLB, and writing the results to the output FIFO; (3) writing the results stored in the output FIFO back to main memory. In this way it is possible, for instance, to use the circuits in the dynamic area to combine information from two images. For high performance, all memory transfers should be managed by the DMA controller.

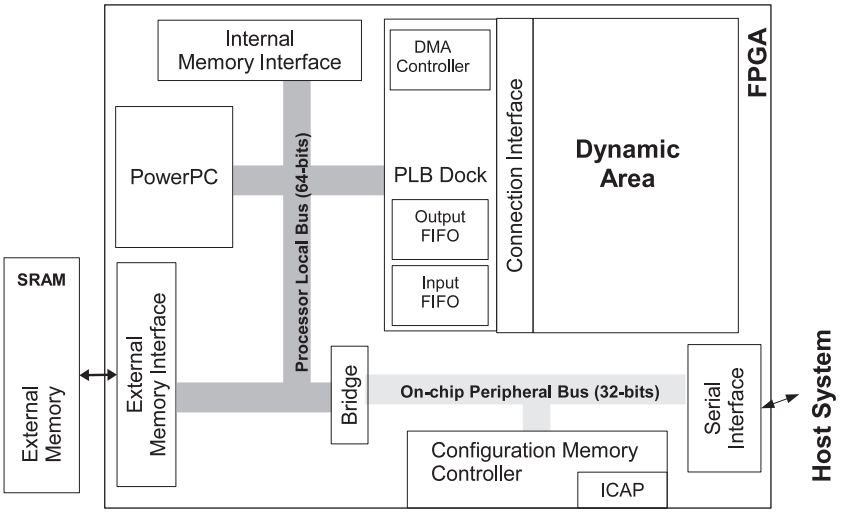

Fig. 1. Architecture of the hardware platform for the prototype implementation. The reserved area for loading partial configurations is shown on the right (the *dynamic area*).

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, and must have a fixed, rectangular footprint. The component's 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, with inputs on the left side and outputs on the right. Each component employs the LUT-based interface macros described in [19].

Components may use dedicated resources like block RAMs and multiplier blocks, but these impose additional restrictions on relocation. Components may also include registers (for instance, in order to implement a pipeline), so accumulators and multiply–add-accumulate units can be used. All sequential components must share the same clock signal provided by the underlying clock distribution network of the FPGA. With these constraints, only the use of embedded blocks affects bitstream generation as described in Section 5.1. The benchmark circuits used in Section 6 use only combinational components that correspond to the basic operations typically supported by SIMD instructions.

The bitstream manipulation tool of [13] 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 (expressed as the number of Configurable Logic Blocks (CLBs) in each dimension), and about the relative positions of input and output terminals. For combinational components information about the worst-case propagation delay from inputs to outputs can also be included. Component description files are grouped together in component libraries. More information about the component design process can be found in [13], which also includes a discussion of the issues related to the physical implementation of the terminals. At run-time, applications can use functions from the software run-time support system to assemble new partial configurations using components from all available libraries. The algorithms implemented for this purpose are described in Section 5.

4. 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 typically 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.

For placement, components are 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 in 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: due to the physical arrangement of the reconfigurable fabric, two adjacent stripes may be separated by an unused BRAM column. The unused BRAM column is considered simply as another set of routing resources.

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) [6]. 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 are not included because they 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 includes the following connections:

- 16 direct connections to the 8 neighboring CLBs (two to each one).
- 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 CLB.
- 32 connections to the inputs of the 4 slices in the CLB.

Fig. 2 shows a simplified view of the switch matrix model (direct connections are not shown). For illustration purposes, the figure shows an hypothetical set of possible internal connections to one output pin (dashed lines). The actual set of connections available is not the same for every output. In addition, some unused output pins can be employed to establish indirect connections (labeled "internal bounce" in the figure). A sample route using two internal connections is also shown. Note that the number of inputs for double and hex lines is twice the number of the corresponding outputs, because each of these lines connects to two neighboring CLBs.

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.

The routing procedure used for connecting components uses the number of segments as the cost of a route. For FPGAs with segmented interconnection architecture, the number of segments has been shown to be a good indicator of the delay [20]. In order to give the designer some feedback about the timing behavior of the generated circuits, the run-time support system estimates the delay of all routes. A delay calculation procedure also provided by the system uses the data to estimate the worst delay through circuits composed of combinational components. The delays associated with each segment type were obtained with help of the FPGA Editor tool (part of Xilinx ISE). In order to have a single delay value for each segment type, we used FPGA Editor to route several lines on different parts of FPGA and with different orientations, and to measure the delays. Since segments of a given type do not always have the same delay, the worst delay for each segment type is used for the estimates.

An example of the setup used to determine segment delays is shown in Fig. 3 for the case of double lines. FPGA Editor does not provide directly the delay values of the segments. However, it can estimate the delay of a route between CLBs. Therefore, it can be used to obtain the delays of routes 1 and 2 of Fig. 3. Since the two routes differ only by one double line and associated switch (the dashed part of route 2), the difference between the corresponding delay values is equal to the delay of the double line. The delays of a segment and of the switch at its beginning are always lumped together in our model.

In some cases, a route goes through more than one switch of the matrix as shown in Fig. 2. In this case, there is an additional delay. Experiments with FPGA Editor showed that the additional delay is (nearly) the same as the one of associated with the segment at the corresponding output of the matrix. The delay estimation procedure takes this effect into account. This approach was used to obtain the following estimated worst-case delay values for the specific FPGA model used in our experimental setup:

- Direct connections and double lines: 0.25 ns.
- Hex lines: 0.34 ns.
- Inputs or outputs to the CLB: 0.15 ns.

These values are in general agreement with the ones reported in [21] for Virtex-II devices. They are valid for the specific model and speed grade used for the measurements; a new set of values must be determined for each different device model and speed grade.

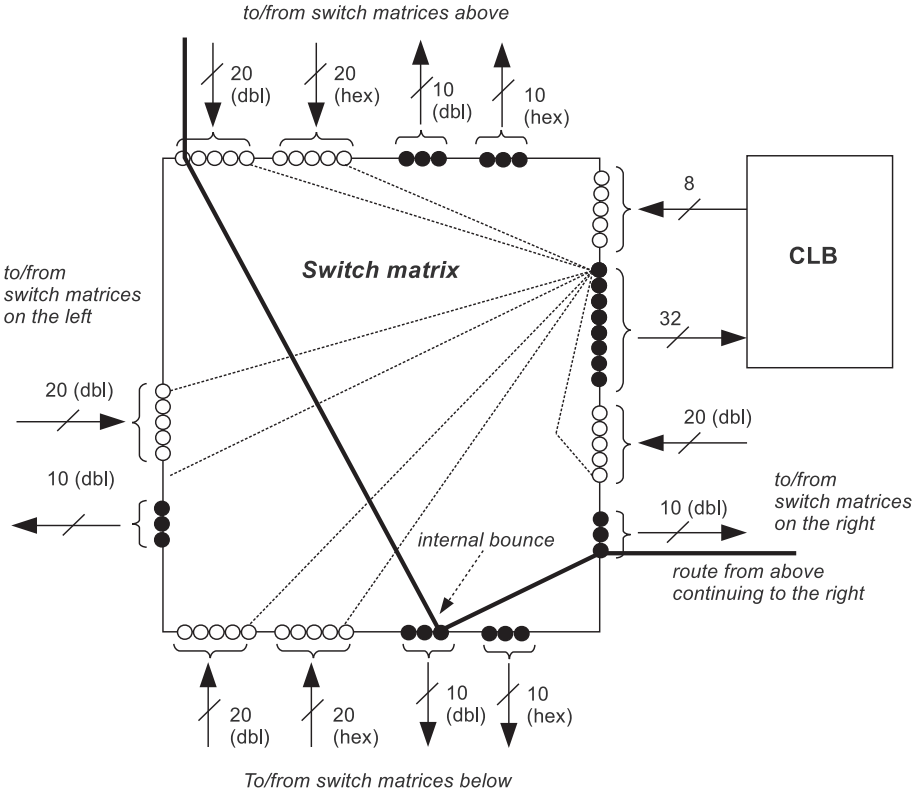

Fig. 2. Simplified representation of the switch matrix model (direct connections not shown). The dashed lines represent a hypothetical set of connections to one specific output. A route going through the switch matrix is shown in bold. Some unused output pins can be used for indirect connections (internal bounce).

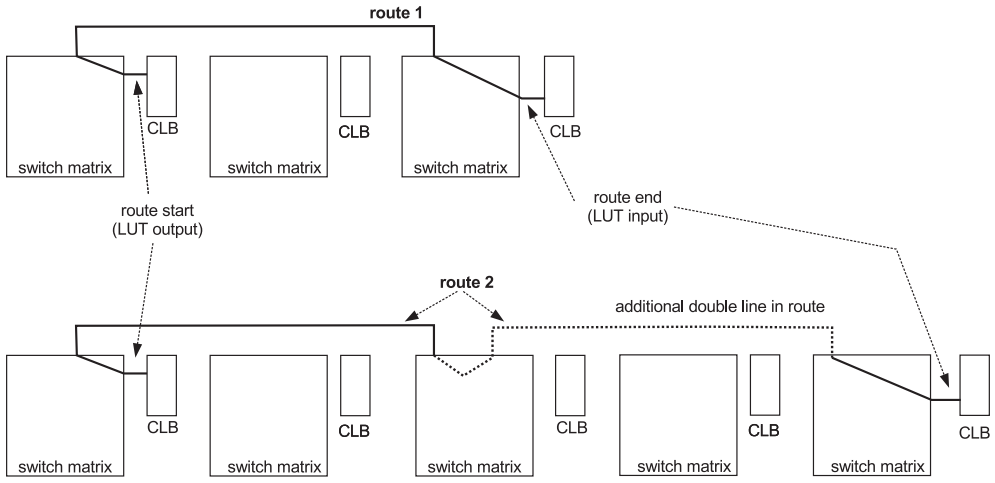

Fig. 3. Setup for obtaining delay information for routing segments.

5. 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).

5.1. 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 Fig. 4.

The placement problem to be solved is similar to the resource constrained scheduling problem of high-level synthesis [22]. Here, the number of usable resources is not constrained by the number of available functional units, but by the height of each stripe and by the total width of all stripes. The solution adopted here follows an as-soon-as-possible (ASAP) strategy.

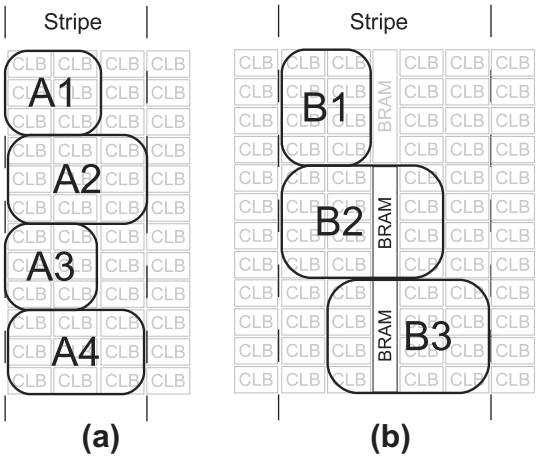

Fig. 4. Placing components in stripes. (a) Typical placement for components that only have CLBs. (b) Placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

The first step in grouping the components is to determine their 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 Fig. 4a, with a three-column stripe). 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. 4b, which shows a three-column stripe stretched to four columns).

The detailed assignment of components to locations processes each level in succession, placing the components from top to bottom. 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. 4b).

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.

5.2. 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 [19], 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 Section 4.

A 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.

In order to determine all the pins involved in a connection, a breadth-first search of the routing area is performed. This strategy has often been used for off-line FPGA routing with fast execution times [23,24]. In our approach, the search is performed at run-time for routing connections between stripes. In addition, the search space is further limited during the search as described next.

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. The area is reduced as the search approaches the target pin, thereby limiting the number of segments 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 [25]. The breadth-first search 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.

Routing is the most time-consuming step of configuration generation. In order to reduce execution time, the router tries to reuse previously calculated routes. For this purpose, it keeps track of the relative positions of endpoints of each route. If it has to process a new connection with the same relative endpoint positions, the router checks whether it is possible to use a vertically shifted version of the previously-found route. If all required resources are free, a new search is avoided. This method exploits the coarse granularity (most operators are 8 or 16 bits wide) and the regularity of the circuits corresponding to sequences of subword operations. After all connections are processed sequentially in this way, a correct partial bitstream is produced from the data stored in the internal data structures.

The implemented 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. Therefore, the order in which connections are processed may influence the final result. (Evaluation of this aspect is outside the scope of the present paper.) The impact of this choice is limited by the fact that routing alternatives are already restricted by the previous placement, and by the design decision to keep any interconnections confined to the area between stripes.

5.3. Using groups of components

The reuse of previously found routes tries to exploit the regularity of the circuits derived from sequences of subword operations. These express naturally the parallel execution of several operations, which ultimately will lead to the use of multiple instances of similar arrangements of operators. In hardware terms, this may lead to the presence of multiple instances of a group of components. The situation is illustrated in Fig. 5a, which shows a circuit containing two instances of a three-component group.

A natural form of exploiting this situation is to place and route a group only once. As happens with route reuse, further occurrences of the same group result in the replication of the previously found solution, as illustrated in Fig. 5b.

The initial prototype implementation has been expanded to handle this situation. For this purpose, groups must be specified in the initial circuit graph. After level assignment, groups are placed and routed individually in a first pass using the approach described previously. The configuration information for each group is shifted to the assigned positions and replicated as necessary. A second pass then places and routes the remaining components using the algorithms described previously.

The use of groups improves the distribution of the components, particularly the alignment between components in contiguous stripes. The improved alignment tends to reduce the size of the routes. The structure of the circuit generated for one of the benchmarks described in the next section (benchmark FE) is shown in Fig. 6: on the left is the version generated by the basic algorithm, and on the right the version generated by grouping two components as in Fig. 5b. The routes between the second and third stripe of Fig. 6a are inside the A1 components of Fig. 6b and are, therefore, much shorter.

6. Experimental evaluation

The algorithms of the previous section were applied to 10 circuit graphs derived from SIMD code fragments. The evaluation was done on a XUP Virtex-II Pro Development System, which has a Xilinx XC2VP30-7 FPGA [6] 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

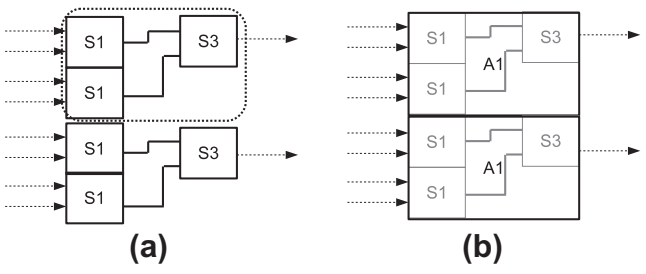

Fig. 5. Circuit specifications obtained from SIMD instructions may have repeated groups of components. (a) Part of circuit graph with two identical groups. (b) Layout built by replicating the layout of one group.

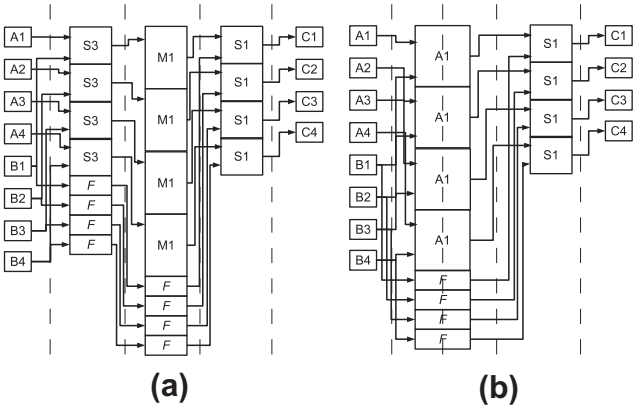

Fig. 6. Component placement for Fade Effect circuit. (a) M1: 8-bit multiplier, S1: 8-bit adder, S3: 8-bit subtractor, Feed: 8-bit feed-through. (b) Placement obtained by grouping S3 and M1 in A1.

version 3.4.1 included in EDK 8.2. The compiled program 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 and come from [3].

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 two versions of the circuit generated for this benchmark are shown in Fig. 6.
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 applied when clipping off triangles for graphics processing. A comparison between four spatial coordinates is performed for three vertexes. The task is the same for each vertex, so vertexes are processed one at a time (from [26]).
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 (from [27]).
9. *Compression (CO):* This task comes from bio-informatics, and consists in compressing a vector of string representation of codes of the four nucleotides into a vector with a 2-bit representation for each nucleotides (from [27]).
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 number (from [27]).

Benchmarks 1–7 come from image processing applications, benchmark 8 occurs in cryptographic applications, and the remaining come from bioinformatics applications. The last two benchmarks involve only bit selection and permutation: they are implemented exclusively by routing, without using any logic blocks.

6.1. Configuration generation time

Table 1 summarizes the structural characteristics of the circuits and the total configuration generation time (including placement, routing, and creation of partial bitstream in the standard device format). The table contains the following information. *In:* number of input bits; *Out:* number of output bits; *Lvl:* number of levels; *Comp:* number of components; *Nets:* number of nets routed; *Bbox:* bounding box (CLB columns by CLB rows); *Time:* time for configuration generation (P: placement, R: routing, B: bitstream creation). Placement includes adding feed-through components to the circuit in order to connect components that are not on successive levels. Routing is executed $L + 1$ times for each circuit (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. All benchmarks required less than 28.2 s total generation time. This is almost completely determined by the routing stage: the most time-consuming placement takes 70 ms.

Table 2 shows the improvements that can be obtained by exploiting the possibility of grouping components. The improved generation procedure can be used with three of the benchmarks. The results show that in those cases where it can be applied, placement and routing using component groups provides a noticeable improvement in generation time. The most time-consuming circuit is processed in 24 s (14.3% improvement) with this alternative.

The segment delay model of Section 4 can be used to estimate the worst-case delay through the generated circuit. For acyclic graphs involving only combinational operators (like the ones used in the benchmarks) a delay may be obtained by doing a breadth-first traversal. When each component is processed, the arrival time of all its input signals is already known, and can be used to calculate when the outputs will be correct. Adding the estimated route delay to the time when outputs of level n are ready determines the input arrival times for the components of level $n + 1$. The delay values obtained in this way provide a conservative worst-case estimate. This is intended as a way of providing the user with some measure of the timing quality of the generated circuits.

The estimated worst-case delay for the benchmark circuits is shown in Table 3. The circuit that has the largest delay corresponds to one of the motion overlay benchmark, which is the one with the largest number of components and nets. The delay of 40.2 ns imposes a maximum clock frequency of 24.8 MHz. Therefore, the circuit can be used without timing problems in the experimental setup, whose dynamic area clock has a maximum frequency of

Table 1
Execution time for configuration generation on the 300 MHz PowerPC 405 embedded in the Virtex-II Pro XC2VP30-7 FPGA.

Circuit	In	Out	Lvl	Components	Nets	BBox ($c \times r$)	Time (s)			
							P	*R*	*B*	Total
BA	32	32	1	4	64	3×12	0.02	8.06	0.006	8.09
CA	32	32	3	12	128	9×32	0.04	16.11	0.04	16.18
AB	64	32	1	4	96	3×12	0.03	12.09	0.01	12.13
FE	64	32	3	12	128	9×32	0.03	16.11	0.04	16.18
MD	64	32	2	8	128	6×12	0.07	16.09	0.02	16.18
MO	64	32	3	16	224	12×28	0.06	28.17	0.08	28.31
CT	32	6	2	7	176	6×15	0.04	22.17	0.03	22.25
LSBS	40	32	2	6	128	6×14	0.05	16.11	0.02	16.18
CO	32	8	1	0	8	3×8	0.001	1.01	0.001	1.01
TR	24	32	1	0	24	3×8	0.002	3.03	0.002	3.03

Table 2
Total configuration generation times for circuits with replicated groups of components.

Circuit	# Groups	Time without grouping (s)	Time with grouping (s)	Improvement (%)
CO	4	16	12	25.0
FE	4	16	14	12.5
MO	8	28	24	14.3

Table 3
Estimated worst-case delays for the benchmark circuits.

Circuit	Delay (ns)
BA	14.6
CA	40.0
AB	13.4
FE	41.2
MD	25.2
MO	40.2
CT	26.1
LSBS	25.6
CO	2.0
TR	2.0

20 MHz. The two fastest circuits (CO and TR) contain no components, so their delay is only due to the interconnections.

6.2. Hardware speedup

The hardware speedup was measured by partially configuring the FPGA with the desired circuit, and then using DMA to transfer data between memory and dynamic area. The generated circuits are all combinational, and produce their results in one clock cycle. The clock of the dynamic area is controlled by the write signal of the PLB bus. In the setup used here, it has an effective maximum frequency of 20 MHz.

For benchmarks that need just one data source, the data is sent directly to the dynamic area and results are accumulated in the output FIFO. Every time the FIFO is full, input transfers are suspended and the results are sent to main memory. When the output FIFO is empty, the next batch of input data is processed.

The benchmarks that combine two images (AB, FE, MD and MO) require two data sources. In this case, the input FIFO is first filled with part of an image; part of the second image is then transferred directly from memory to the dynamic area. The dynamic area receives (on each cycle) data from the input FIFO and from main memory, and stores the results in the output FIFO. When the output FIFO is full (and the input FIFO empty), the results are sent to main memory, and the process is repeated until the complete images have been processed.

In order to compare the performance of benchmarks requiring one and two images, execution time has been normalized by dividing the time by the number of output pixels produced (Table 4). For the benchmarks that are not taken from image processing applications, Table 5 reports the execution time for 250,000 calculations.

Tables 4 and 5 compare the running times of software and run-time-generated hardware versions of the benchmarks. The software binaries are produced with the GNU C compiler version 3.4.1 (from EDK 8.2) using the O2 optimization level. The hardware execution times are, in these cases, determined only by the data transfer times (DMA setup time included). In Table 4 they are different for tasks that process one image (BA, CA) or two images. All tasks in Table 5 involve the same amount of data transfers and therefore have the same execution time.

The speedups 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. If partial configurations are reused during the same application, overall performance may be improved by keeping a cache of previously-generated configurations. The implementation of more complex computations in hardware would allow more operations to be executed for the same time spent in data transfers to/from the dynamic area, possibly improving the observed speed-ups.

The results described in this section were obtained with a Virtex-II Pro device. This approach is extensible to other device

Table 4
Execution time per output pixel for image processing tasks. All images have 8-bit pixels (gray-scale) and size 1024×1024.

Tasks	Software (μ s)	Hardware (μ s)	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

Table 5
Execution time for non-image-processing benchmarks. Each task is executed 250,000 times.

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

families based on a rectangular array of CLBs connected by a segmented interconnection network. Differences in the type of embedded blocks available or in the size of the LUTs have little impact, since components are treated as black boxes. Changes in the interconnection network would have a larger effect, but mainly restricted to the routing resource model described in Section 4.

Changes in the bitstream format can have a larger impact. For instance, more recent device families have frames with a standard height corresponding to a clock region [7] (which is smaller than the device height). This change makes it easier to reconfigure smaller areas of the device without affecting others, and leads to shorter reconfiguration times. Placement and routing procedures would have to be altered to take explicit advantage of the finer-grained bitstream organization.

7. 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. Configuration generation time is reduced by (a) reusing information from previously-routed connections, (b) reusing information from groups of components that are placed and routed in a preliminary pass.

For a set of ten SIMD-code-derived benchmarks, the generation times varied between 1 s and 24 s; the generated hardware exhibited speedups between 11 and 73 compared to the software versions.

The evaluation of the suitability of this approach for specific cases 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. Scenarios that may accommodate delays in the range under discussion 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 are able to amortize the cost of generation by caching the newly-created configurations for reuse. Another scenario involves adaptive systems that use learning (for instance, of the number of taps and the coefficients of a FIR filter) 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.

For these situations the working implementation described here shows that run-time generation of configurations is a feasible technique for use in embedded systems implementing computationally-demanding multimedia and digital signal processing tasks.

Acknowlegement

This work was partially supported by research contract PTDC/EEA-ELC/69394/2006 from the Foundation for Science and Technology (FCT), Portugal.

References

[1] J. Gause, P. Cheung, W. Luk, Reconfigurable computing for shape-adaptive video processing, IEE Proceedings – Computers and Digital Techniques 151 (5) (2004) 313–320.
[2] K. Paulsson, M. Hiibner, J. Becker, J. Philippe, 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.
[3] A. Peleg, S. Wilkie, U. Weiser, Intel MMX for multimedia PCs, Communications of the ACM 40 (1) (1997) 24–38.
[4] Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1. Basic Architecture, October 2011.
[5] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scales, AltiVec extension to PowerPC accelerates media processing, IEEE Micro 20 (2) (2000) 85–95.
[6] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, version 4.7, November 2007.
[7] Xilinx, Virtex-5 FPGA Configuration User Guide, version 3.9, August 2010.
[8] ATMEL, AT94KAL Series Field Programmable System Level Integrated Circuit, January 2008.
[9] P. Lysaght, B. Blodget, J. Mason, J. Young, B. Bridgford, Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL 2006), 2006, pp. 1–6.
[10] E.L. Horta, J.W. Lockwood, D.E. Taylor, D. Parlour, Dynamic hardware plugins in an FPGA with partial run-time reconfiguration, in: Proceedings of 39th Design Automation Conference, 2002, pp. 343–348.
[11] Y. Krasteva, E. de la Torre, T. Riesgo, D. Joly, Virtex II FPGA bitstream manipulation: application to reconfiguration control systems, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL 2006), 2006, pp. 1–4.
[12] H. Kalte, M. Porrmann, REPLICA2Pro: task relocation by bitstream manipulation in Virtex-II/Pro FPGAs, Proceedings of the 3rd Conference on Computing Frontiers, ACM, 2006, pp. 403–412.
[13] M.L. Silva, J.C. Ferreira, Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems, IET Computers & Digital Techniques 1 (5) (2007) 461–471.
[14] J. Aycock, A brief history of just-in-time, ACM Computing Surveys 35 (2) (2003) 97–113.
[15] M. Arnold, S.J. Fink, D. Grove, M. Hind, P.F. Sweeney, A survey of adaptive optimization in virtual machines, Proceedings of the IEEE 93 (2) (2005) 449–466.
[16] E. Bergeron, M. Feeley, J.P. David, Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs, in: Proceedings of the Joint European Conferences on Theory and Practice of Software/17th International Conference on Compiler Construction, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 178–192.
[17] J. Suris, C. Patterson, P. Athanas, An efficient run-time router for connecting modules in FPGAS, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL 2008), 2008, pp. 125–130.
[18] M.L. Silva, J.C. Ferreira, Generation of partial FPGA configurations at run-time, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL 2008), 2008, pp. 367–372.
[19] M. Hnbner, T. Becker, J. Becker, Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration, in: Proceedings of 17th Symposium on Integrated Circuits and Systems Design (SBCCI 2004), 2004, pp. 28–32.
[20] M. Wang, A. Ranjan, S. Raje, Multi-million gate FPGA physical design challenges, Proceedings of 2003 IEEE/ACM International Conference on Computer-Aided Design, IEEE Computer Society, 2003, pp. 891–898.
[21] K. Paulsson, M. Huebner, J. Becker, Exploitation of dynamic and partial hardware reconfiguration for on-line power/performance optimization, in: International Conference on Field Programmable Logic and Applications, 2008, 2008, pp. 699–700.
[22] R.A. Walker, S. Chaudhuri, Introduction to the scheduling problem, IEEE Design & Test of Computers 12 (2) (1995) 60–69.
[23] D.D. Hill, A CAD system for the design of field programmable gate arrays, Proceedings of 28th ACM/IEEE Design Automation Conference, ACM, 1991, pp. 187–192.
[24] J.S. Swartz, V. Betz, J. Rose, A fast routability-driven router for FPGAs, Proceedings of 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, ACM, 1998, pp. 140–149.
[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., McGraw-Hill, 2003.
[26] R.B. Lee, A.M. Fiskiran, PLX: an instruction set architecture and testbed for multimedia information processing, The Journal of VLSI Signal Processing 40 (1) (2005) 85–108.
[27] Y. Hilewitz, R. Lee, Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors, Journal of Signal Processing Systems 53 (1) (2008) 145–169.

"**Miguel L. Silva** received his BS in Electrical and Computer Engineering from the Faculty of Engineering of the University of Porto and a MS in Artificial Intelligence and Computation from the same University. He is currently a PhD student in Electrical and Computer Engineering at the Faculty of Engineering of the University of Porto. His research interests include dynamic reconfigurable systems, FPGA's, configurable resource management and CAD tools."

"**João Canas Ferreira** received the Ph.D. degree in electrical and computer engineering from the University of Porto (Portugal), in 2001. He is currently an assistant professor with the Faculty of Engineering, University of Porto. His current research interests include dynamically reconfigurable systems, application-specific digital system architectures, and ECAD tools and algorithms."