Algorithms for run-time placement and routing on Virtex II Pro FPGAs[*]

Miguel L. Silva
DEEC, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias,
4200-465 PORTO, Portugal
mlms@fe.up.pt

João Canas Ferreira
INESC Porto, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias,
4200-465 PORTO, Portugal
jcf@fe.up.pt

Abstract

Run-time reconfiguration is a useful approach to the implementation of highly-adaptive embedded systems. To generate partial bitstreams at run-time for dynamic reconfiguration of sections of a platform FPGA we combine partial bitstreams of coarse-grained components specified by an acyclic netlist. The placement an routing algorithm play and essential role on the generation of partial bitstreams. A greedy placement heuristic based on topological sorting is used to determine the positions of individual components, and a router based on non-backtracking search over restricted areas determines the routes for the interconnections. The approach is validated with a set of 35 benchmarks (both synthetic and application-derived) having between three and 41 components, the complete process of bitstream generation takes between 7 s and 101 s (average 48.3 s) when running on an embedded PowerPC 405 microprocessor clocked at 300 MHz.

1. Introduction

This paper proposes a method to generate partial bitstreams at run-time in order to partially reconfigure an FPGA. The hardware infrastructure is assumed to include a microprocessor for running the bitstream creation procedure, and to load the newly created bitstream to a specific FPGA area without disturbing the operation of other parts of the system.

For the specific implementation described here, the program runs on an embedded processor in the same FPGA that is being reconfigured. Highly adaptive embedded systems may employ the creation of partial bitstreams at run-time in situations where it is impractical to create all necessary bitstreams at design time, either because there are too many possibilities (e.g., shape-adaptive video processing [1]), or because the required information is only available at run-time (e.g., self-adaptive systems [2]).

The proposed approach is based on placing medium-sized components (like adders, comparators, and multipliers) in a reserved area, and then routing the interconnections among the components, and between the components and the area's I/O terminals. Since platform FPGAs have a heterogeneous fabric (with, e.g., RAM blocks and dedicated multipliers), information about the relative position of resources in the component is required to determine whether a specific location is compatible. For routing purposes, components are treated as black boxes with I/O pins at the periphery. The final partial bitstream is created by merging the component bitstreams (after relocation) into the bitstream for the empty reserved area, and then by further modifying the result to include the connections determined in the routing phase.

Because placement and routing must be performed in a resource-limited context, simple algorithms are employed with the purpose of obtaining acceptable solutions in a reasonable time. Placement is done by a greedy strategy based on sorting the components in topological order. Routing is performed by finding the shortest path from a source terminal to the target terminal for successive nets; target terminals belonging to the same net can share routing resources. These procedures have been implemented in the C programming language and included in a code library for use by applications that wish to take advantage of this approach to improve system adaptability without foregoing hardware support for compute-intensive routines.

An implementation of the proposed approach was evaluated for synthetic and application-derived benchmarks containing between three and 41 components (average: 15 components). For this set of benchmarks, the whole process of bitstream generation takes between 7 s and 101 s (average 48.3 s) on a PowerPC 405 microprocessor clocked at 300 MHz. Both the hardware organization and the process for component creation process employed in the prototype that was used to collect these results have been previously described in [3]. The hardware platform has a Xilinx Virtex-II Pro device with two embedded PowerPC cores [4], although only one is used for this work. The system has a reserved dynamic area that can be configured through partial bitstreams that are loaded using the internal configuration access port (ICAP).

The rest of the paper is organized as follows. Section 2 describes the context for the research reported in the paper and describes previous work. Section 3 presents the details of the placement and routing tasks, including the simplified resource model adopted in order to reduce execution time.

[*]First author funded by FCT scholarship SFRH/BD/17029/2004. Work supported by FCT project PTDC/EEA-ELC/69394/2006.

Results for an implementation running on the Virtex-II Pro platform FPGA are reported in section 4 for a set of 35 benchmarks. Finally, section 5 presents the conclusions.

2. Related work

In the context of FPGA-based systems, run-time reconfiguration (RTR) designates the capability to alter the hardware design realized by the FPGA in the course of the execution of an application. Basic RTR requires just fast reconfigurability, typically provided by an SRAM-based FPGA. More effective use of RTR can be made if the FPGA supports partial active reconfiguration, i.e., when sections of the reconfigurable fabric may be re-programmed without affecting other sections. This feature enables the implementation of compact, self-adapting systems.

One of the issues raised by RTR concerns the generation of the required partial configurations. This is commonly done at design time, when all eventually useful partial configurations must be specified and created [5, 6]. Several approaches to the relocation of partial bitstreams have been proposed, including both software tools [7,8] and hardware solutions [9,10]. Bitstream relocation is explicitly included in recent design flows [11].

In all cases, the synthesis tools must be run for each partial configuration, making the generation of partial configurations time-consuming. A solution to this problem based on building new partial bitstreams by combining bitstreams of smaller components is described in [3]. The creation of the new bitstreams requires assigning positions of the reconfigurable area to components (placement), relocating and merging the individual component bitstreams, and interconnecting the components (routing) by modification of the merged bitstream. Because this approach does not rely on the synthesis of logic descriptions, it is a good candidate for implementation in an embedded system for the purpose of creating new dynamically reconfigurable modules (partial bitstreams) at run-time.

Efforts to speed-up placement and routing for FPGAs were initially motivated by applications to logic emulation and custom computing. Trade-offs between area and execution time for placement are discussed in [12], where the authors describe a placer that obtains a 52-fold reduction in execution time for a 33% increase in circuit area. Trade-offs between execution time and critical path delay for placement and routing of FPGA circuits are discussed in [13] . By combining different algorithms for placement and routing, the authors of that work obtained a wide range of solutions, including a 3-fold speedup with a 27% degradation of critical path delay. A router for just-in-time mapping of a device-independent configuration description to a specific device architecture is described in [14] : that router is able to produce good hardware circuits using 13 times less memory and executing 10 times faster than VPR [15].

A channel router for the Wires-on-Demand RTR framework is implemented in [16] . The router uses a simplified resource database that is several orders of magnitude smaller than the one used by vendor tools. It uses simple algorithms to find local routes between blocks using relatively few computational resources. Results obtained with a 2.8 MHz 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).

A simplified version of the bitstream assembly approach of [3] is implemented in [17] for an embedded system with a Virtex-II Pro device. The system used described and evaluated in the next sections is an evolution and extension of that work.

3. Placement and routing

Placement and especially routing are generally very demanding tasks. In order to perform them at run-time in embedded systems, we work with coarse-grained components, and use a simplified model of the resources together with simple, greedy place and route algorithms. The main goal is to find a useful solution rapidly, not to exploit all the available resources optimally.

3.1. Resource models

Placement and routing for island-style FPGAs like the Virtex-II Pro is a resource and time consuming task, in part due to the need to handle a large amount of fine-grained resources. For an embedded system with limited computational resources a more coarse-grained approach is required.

In the approach presented here the basic functional element is a component that takes up a certain area of the FPGA fabric (specified in CLBs). This rectangular-shaped component must have all its terminals on the left or right sides. Physically, the terminals are inputs or outputs of LUTs defined at design time. Typically, components have a functional core between a left column of input CLBs and a right column of output CLBs, although this arrangement is not strictly necessary. Terminals must be located on the borders, because the components are considered as black boxes during placement and routing: no overlap of components is allowed and no routing over components is done.

The simplified placement procedure groups components into vertical stripes. The position of a component inside a stripe and the width of the stripe depend on the physical resources used by the component. Routing is restricted to connections between components in adjacent stripes. This restriction guarantees that routing does not interfere with the rest of the system, reduces the search space, and simplifies the process significantly.

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, to account for constraints imposed by the embedded block RAMs (BRAMs).

Due to the physical arrangement of the reconfigurable fabric, two adjacent stripes may be separated by an unused BRAM column in some cases. The unused BRAM column is considered simply as another set of routing resources.

The Virtex-II Pro FPGA has a segmented interconnection architecture, where segments are connected by a regular array of switch matrices, which are connected between themselves and to the other resources (like CLBs and BRAMs) [18]. A large number of routing resources, grouped in vertical and horizontal channels, connect the switch matrices. In order to simplify routing, only a subset of the available segments is used:

- 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 (i.e., bidirectional wires that distribute signals across the full device height and width) are excluded, because they can interfere with circuitry outside of the dynamic area. Horizontal hex lines were excluded because they connect to every third or sixth CLB to the left or the right, and therefore reach beyond the area reserved for routing. It is unnecessary to consider other dedicated routing resources (like carry chains, for instance), because they have no bearing on the connections that are to be established at run-time.

The resulting simplified model of the switch matrix associated with each CLB 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.

A switch matrix pin is identified by its index in this list of pins. It is also necessary to keep information on the possible connections from a given pin to other pins of the switch matrix. The information required in this case includes the following data for each target pin:

- identification (index) of the target pin;
- relative vertical distance of the endpoints of the connection starting at the target pin (e.g., +1 and +2 in the case of a double line connection in the up direction);
- relative horizontal distance of the endpoints of the connection starting at the target pin (e.g., -1 and -2 in the case of a double line connection in the left direction).

The algorithm of section 3.3 models the area reserved for routing 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.

Algorithm 1: Greedy level-oriented component placement

```
Data: Netlist $N$ of all components
Result: $B$: merged bitstream with all components
        $R$: routing information

$L \leftarrow$ LevelAssignment($N$)
AddStripeInformation($L$)
$x \leftarrow 0, \ell \leftarrow 0$
initialize B to the default bitstream
initialize R
foreach $S \in L$ do
    $y \leftarrow 0$
    foreach $c \in S$ do                              // ordered scan of $S$
        $y_1 \leftarrow y +$ YOffset($c$)
        if $y_1 +$ Height($c$) $\leq$ DeviceHeight then
            $x_1 \leftarrow x +$ XOffset($c$)
            merge bitstream of component c into B at $(x_1, y_1)$
            if
            Width($c$) + XOffset($c$) < Width($S$) $\vee x_1 \neq x \vee y_1 \neq y$
            then
                merge feed-through components in B
            update R with final terminal positions for inserted
            component
            $y \leftarrow y_1 +$ Height($c$)
        else
            fail
        if max(Level(Successors($c$))) $> \ell + 1$ then
            insert feed-through component in level $\ell + 1$ of N
    $\ell \leftarrow \ell + 1$
    $x \leftarrow x +$ Width($S$)
```

3.2. Placement

The main input to the placement phase is a component netlist specifying 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 general approach to placement is to find an arrangement of components in columns, so that directly connected components are adjacent to each other. The arrangement in columns was chosen because it matches the reconfiguration mechanism of Virtex-II-Pro FPGAs, where the smallest unit of reconfiguration data (called a frame) applies to an entire column of resources.

A high level description of the implemented greedy placement approach is shown in Algorithm 1, and two examples of component placement inside a stripe are displayed in figure 1. Positions are specified in terms of CLB rows and columns, with the origin at the top left corner of the device.

The first step of the placement algorithm is to group the components by levels (function `LevelAssignment()`. The first level contains the components whose inputs are connected to the interface of the dynamic area. Second level components have all their input terminals connected to first-level components and so forth. If a component has more than one source, the component will be assigned to the level following the highest-numbered source. This is equivalent to processing the components in topological order.

The next step is to determine the set of contiguous CLB columns (a stripe) required for all components of each level (call to `AddStripeInformation`). The final placement of a component will be restricted to the columns assigned to its level. The starting column assigned to a given

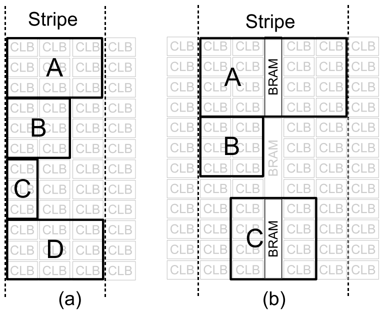

Figure 1. Placing components in stripes. (a) Typical placement for components that only have CLBs; (b) Placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

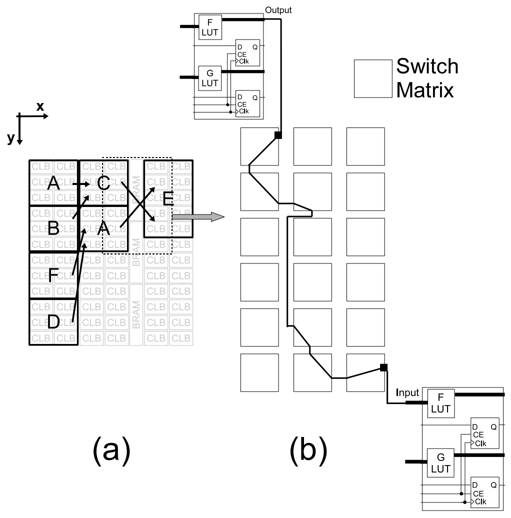

Figure 2. Routing between stripes. (a) Placed components with indication of connections to be established; (b) detail of routing area (dashed box) showing one possible route connecting C to E.

level will be the one closest to the dynamic area interface without overlapping columns of previous levels. The number of columns assigned to a stripe is the smallest required to accommodate all components of the corresponding level (see Figure 1a). 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 Figure 1b).

Placement proceeds by processing each level in succession and placing the components from top to bottom of the device. If possible, a new component is placed just below the previous one ($\texttt{YOffset}(c) = 0$) and at the right edge of the stripe ($\texttt{XOffset}(c) = 0$). However, the placement of components with non-homogeneous resources (like BRAMs) may require offsetting the component from the default location ($\texttt{YOffset}(c) > 0$ or $\texttt{YOffset}(c) > 0$). As a result, components may not start at the left edge of the stripe, nor end at the right edge. In all cases, the empty spaces in the stripe are filled with feed-through components, ensuring that all outputs are brought to the right side of stripe.

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. The placer generates all feed-through components as required, without recourse to library components.

Placement 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 device. At the end of the placement procedure the information on the final positions of all component terminals is collected for use in the routing stage.

The automatic placement strategy assumes that components have input terminals on their left border and output terminals on the right. As an alternative to automatic placement, the run-time support library contains functions that allow the explicit placement of components by the application. In this case, both types of terminals may be present on either edge of the routing area. For both types of placement, a list of connections and associated physical terminal positions is created for use as input to the router.

The final result of a successful placement consists of the default partial bitstream merged with the relocated bitstreams of the components.

3.3. Routing

The routing procedure described in this section is used to establish connections between terminals of components in adjacent stripes. The procedure implements a breadth-first search of the routing area, which is represented by an array of switch matrices, one for each CLB in the area. 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 column between the stripes.

Physically, component terminals are pins of the switch matrix of the corresponding CLB. The component inputs connect to the input pins of the CLB LUTs, while component outputs connect to the slice outputs [18]. Other pins in the switch matrix connect to corresponding pins in other switch matrices. So the result of routing one net is simply the set of switch matrix pins required to establish the desired connectivity, which implicitly define the settings of the switch matrices involved. The situation is illustrated in Figure 2.

The actual area searched starts as the smallest rectangle of switch matrices that encloses all pins used as terminals of the net to be routed, and is reduced during the search. Since the search area is restricted, the number of possible connections to be examined is limited. Restricting the search area in this way reduces the chances of successfully routing a given netlist, but reduce the search effort signifi-

Algorithm 2: Greedy breadth-first routing algorithm

```
Data: List $R$ of nets to route
      Partial bitstream $B$ with components
      Search area $A$
Result: Partial bitstream $B$ with merged routes
1  usedPins ← create switch matrix array for search area A
2  foreach $n \in R$ do
3      currentPins ← {GetNetSource($n$)}
4      destinationPins ← GetNetSinks($n$)
5      distLimit ← max(Width($A$),Height($A$))
6      solutionPaths[$n$] ← $\emptyset$
7      newCurrentPins ← $\emptyset$
8      $f$ ← SelectOne(destinationPins)
9      while |Reached| < |destinationPins| do
10         if currentPins = $\emptyset$ then fail
11         allPins ← set of pins connected to any element of currentPins
12         foreach $p \in$ allPins do
13             if $p \in$ Visited then continue
14             Visited ← Visited ∩ {$p$}
15             $d$ ← Distance($p, f$)
16             if $p \notin$ usedPins ∧ InSearchArea($p$) ∧ $d \le$ distLimit
               then
17                 if $p \in$ destinationPins then
18                     newPath ← RetracePathTo($p$)
19                     MergePaths(solutionPaths[$n$], newPath)
20                     Reached ← Reached ∩ {$p$}
21                     if $p = f$ then
22                         $f$ ←
                           SelectOne(destinationPins \ Reached)
23                         Visited ← $\emptyset$
24                         newCurrentPins ←
                           {GetNetSource($n$)}
25                         distLimit ←
                           max(Width($A$), Height($A$))
26                         break
27                 else
                       // it is not an endpoint
28                     newCurrentPins ← newCurrentPins ∩ {$p$}
29                     distLimit ← min(distLimit, $d$)
30         currentPins ← newCurrentPins
31     add all pins from paths in SolutionPaths[n] to usedPins
32     clear all flags, Visited ← $\emptyset$, Reached ← $\emptyset$
33 configure bitstream with all elements of solutionPaths[]
```

cantly. The restricted routing algorithm remains capable of routing significant classes of circuits, as shown empirically in section 4.

The high-level description of the routing algorithm for one region is shown in Algorithm 2. The algorithm performs a breadth-first search for a shortest-path forest between the source of a net (one component's output terminal) and its sinks (one or more input terminals). Nets are processed in sequence, without reconsidering the routing of previous nets (outer loop in line 2).

During the search, variable currentPins contains the pins that belong to the border of the expanding search, i.e., those pins that could be reached from the source in the number of steps corresponding to the number of iterations of the inner loop starting (line 9). Initially, only the source of the net belongs to this set (line 3); during execution, the successors of each visited pin are added (lines 28)

The loop at line 9 is repeated until every sink is reached. The distance of a pin p to the current target sink f is used to limit the search at line 16. The function Distance(p,f) (used in line 15) is equal to the largest of the vertical and horizontal distances between p and f: Distance(p,f) = $\mathtt{max}(|x_p - x_f|, |y_p - y_f|)$. Only pins within a distance distLimit are eligible for consideration. The value of distLimit is initialized to the largest dimension of the search area (generally, its height) and reduced as the search progresses (line 29). The variable is reset to the initial value after reaching each sink.

The variable newCurrentPins holds the set of pins to be used as starting points in the next iteration of the search. This set includes all pins directly connected to the pins in currentPins that have not yet been visited in the course of this search. Every pin added to newCurrentPins includes a reference to its predecessor on the search path. As the search is extended to neighboring pins, these are flagged as "visited", to avoid repeated processing and ensure that only shortest paths are considered.

Every time an element of destinationPins is reached, a path is created by retracing through the chain of predecessor pins (function RetracePath()). On reaching the current sink target, the search for any remaining endpoints of the same net is setup (line 21): newCurrentPins again contains only the source pin, Visited is now a empty set, and a new target sink is selected from the remaining ones. Once all sinks have been found (line 31), pins used in the solution are added to usedPins, state information for the current search is reset and the next net is processed. The final step updates the bitstream with the configuration information for the new routes.

The algorithm presented here does not ensure that a global optimum for all routes is obtained, since each net is treated in isolation, without considering the impact on the following nets. In addition, the dynamic restriction of the search area (for performance reasons) may cause some solutions to be ignored. The current implementation does not try to adjust the order in which nets are processed and does not control the congestion of the routing area during the search. The impact of these limitations is mitigated by the fact that the router's choices are considerably restricted by the previous placement, and by the design decision to keep any routing-related modifications confined to a relatively small inter-component area. As shown by the benchmark circuits of section 4, a large variety of circuits can be successfully routed by this approach.

4. Experimental results

The performance of the algorithms of section 3.3 was evaluated by applying them to a set of benchmark circuits. The evaluation was done on a XUP Virtex-II Pro Development System, which uses 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 for this work. The CPU operates at 300 MHz, and the 64-bit processor local bus connected to the memory controller employs a 100 MHz bus clock.

This section presents the results obtained by applying the placement and routing algorithms to three sets of benchmarks. For both sets, component dimensions and terminal positions have realistic values derived from actual designs.

The first set of benchmarks comprises four classes of

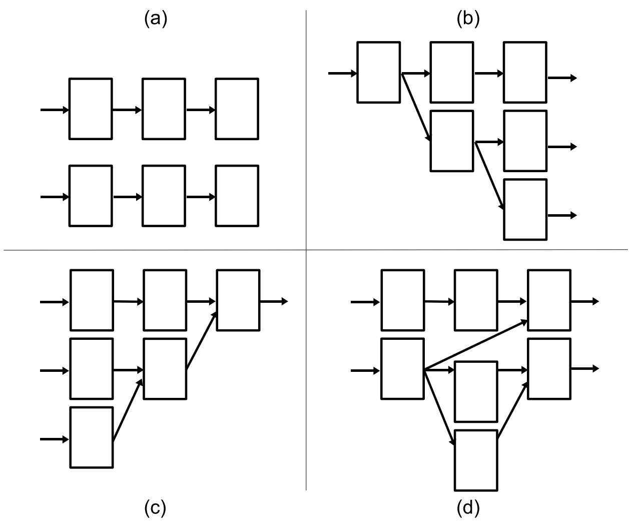

Figure 3. Example of circuit graphs for each class of the first set of benchmarks.

	Number Inputs	Number Outputs	Levels	Number of Modules 8-bit	16-bit	32-bit	Nets	Maximum fan-out
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	2	288	4

Table 1. Basic structural characteristics of all example circuits from the first set of synthetic benchmarks.

synthetic circuits, whose general structure is depicted in Figure 3:

Pipeline (a) One or more pipelines;

Tree SM (b) Tree-like graphs with a single input component and multiple output components;

Tree MS (c) Tree-like graphs with multiple input components and a single output component;

Random DAG (d) Random directed acyclic graphs.

The structure of the first three classes is well matched to the behavior of the placement algorithm, while the last class is more general.

Table 1 describes the basic characteristics of the individual examples: the number of input and output ports, the number of components working with each of the three different data sizes (8, 16 and 32 bits), the number of levels of the structure, and the maximum fan-out (number of sinks of a net).

The other two sets of benchmarks are an adaptation of benchmarks used by [19]:

Random binary expressions This set consists of 6 random binary expressions, which produce a binary tree structure, whose leaf nodes are the input constants and the root node is the result of the expression. All internal nodes are binary operations. The structural details of each benchmark are summarized in Table 2.

Honeywell/MediaBench The last set is based on nine data flow graphs adapted by [19] from the Honeywell [20] and MediaBench benchmarks [21]. All nodes are assumed to process 8-bit data items. Table 3 shows the structural details of all circuits from this set.

For the complete set of benchmarks, the average number of components is 15 and the average number of connections is 164.

	Number Inputs	Number Outputs	Levels	Number of 8-bit Modules	Nets
tg01	40	8	4	9	72
tg02	80	8	5	19	152
tg03	48	8	4	11	88
tg04	80	8	6	19	152
tg05	80	8	5	19	152
tg06	120	8	5	25	200

Table 2. Basic structural characteristics of the second set of benchmarks representing binary random expressions. The names of the benchmarks are the ones used in [19].

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.

Table 4 summarizes the results of running the place and route algorithms on the benchmark circuits. For each benchmark, Table 4 presents the total time required for bitstream generation, the number of levels of the corresponding graph, 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 those used for feed-through routing. The last column shows the relative area occupied by feed-

	Number Inputs	Number Outputs	Levels	Number of 8-bit Modules	Nets
Honeywell-intfc01	24	40	4	13	104
Dft	56	32	3	12	64
Honeywell-versatil01	24	40	5	20	144
Honeywell-intfc02	48	56	7	21	152
Honeywell-fft01	40	64	6	23	168
Honeywell-fft02	56	56	7	24	184
MediaBench-jpeg	56	128	5	27	200
Honeywell-fft03	48	32	8	28	240
Honeywell-versatil02	72	104	8	41	328

Table 3. Basic structural characteristics of the circuits from the Honeywell and MediaBench benchmarks (adapted from [19]).

	Time (s)	Bounding box (Columns x Rows)	Number of feedthroughs	Total component area (CLBs)	Area used for feedthoughs (%)
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	92.43	24x32	30	200	13
tg01	18.84	10x11	0	71	0
tg02	57.78	13x17	0	151	0
tg03	22.03	10x18	0	96	0
tg04	54.87	16x17	0	166	0
tg05	55.69	13x17	0	151	0
tg06	66.94	13x27	0	211	0
Honeywell-intfc01	20.86	21x16	18	156	10
Dft	13.62	9x28	0	144	0
Honeywell-versatil01	40.58	18x29	24	240	9
Honeywell-intfc02	44.14	21x29	15	252	6
Honeywell-fft01	45.62	18x33	6	276	2
Honeywell-fft02	62.91	21x32	24	288	8
MediaBench-jpeg	53.16	15x32	18	324	5
Honeywell-fft03	81.67	24x35	36	336	10
Honeywell-versatil02	100.73	24x56	15	492	3

Table 4. Results of the execution of the placement and routing algorithms on the 300 MHz PowerPC 405 embedded in the XC2VP30-7 FPGA.

through components.

The running time is completely determined by the routing stage. The most time-consuming placement took only 154 ms for the Honeywell-fft03 benchmark. The number of levels L is equal to the number of stripes. Therefore, the routing procedure (Algorithm 2) is called $L+1$ times for each benchmark, for connections between strips and connections for the input and outputs. For Virtex-II-pro FPGAs the size of the partial bitstream, and therefore the time taking for partial reconfiguration, is proportional to the number of columns occupied by the circuit (first number in the fourth column). The typical dynamic area of our test system is 22 columns by 32 rows. Most of the benchmarks fit this reserved area; the four that do not, still fit comfortably our target FPGA, which has 46 columns by 80 rows.

Routing may involve adding feed-through components to the circuit in order to connect components that are not on successive levels. With the exception of two benchmarks (the two largest random DAGs), the additional components do not represent more 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), the largest random DAG and the largest Honeywell benchmark. The global average running time is 48.3 s. These running times make the current version unsuitable for applications that require a very fast turnaround time, like just-in-time compilation.

For the hardware setup used in this evaluation, a one-time reduction in running time might be obtained by using both CPU cores: since the routing area between stripes can be processed independently, routing may be easily performed concurrently by both processors. Another possibility, applicable to situations in which partial configurations are reused during the same application run, is to maintain a configuration cache.

There are, in addition, 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.

Another application involves self-diagnosis of malfunctioning systems. In this case, normal operation has not yet begun (or has been interrupted). Depending on the results of some initial self-tests, the system may proceed to a diagnosis phase, during which new test hardware is generated which depends on the results of the previous tests. In this case, run-time generation would avoid the need to pre-generate and store a potentially very large number of specific diagnostic circuits (most of which would never be used).

The current system is also useful in adapting components to a design-specific dynamic area interface. Often, it is desirable to re-use some (large) component in several systems having different configurations of the dynamic area (in particular, the position of the connections between the dynamic and static areas may change). The component might even be a third-party intellectual property block, designed without any knowledge of the physical details of the dynamic area. With the current system, the physical interface adaptation might be performed at run-time by routing the appropriate connections between the reserved area interface and the component.

5. Conclusion

This paper presents the first implementation and evaluation of an embedded system that is able to generate partial bitstreams at run-time for use in the dynamic reconfiguration of sections of a Virtex-II Pro platform FPGA. The goal is to obtain useful solutions in a short time. The system uses a greedy placement heuristic based on topological sorting to determine the positions of individual coarse-grained components whose interconnections are specified by an acyclic netlist. A router based on non-backtracking search over restricted areas determines the routes for the interconnections. The partial bitstream is constructed by merging together a default bitstream of the reconfigurable area, the relocated partial bitstreams of the components, and the configuration of the switch matrices used for routing. The computational effort is kept bounded by a combination of factors: circuit description by acyclic netlists of coarse-grained components, simplified resource models, direct placement procedure, and the restricting of routing to limited areas.

The results for a set of 35 benchmarks (both synthetic and application-derived) show that time required for bitstream generation on a 300 MHz PowerPC embedded processor depends strongly on the complexity of the circuits, averaging 48.3 s (minimum: 6.97 s, maximum: 100.73 s) for an average circuit size of 15 components (minimum: 3, maximum: 41) and 164 connections (minimum: 32, maximum: 328).

The working implementation described here shows that run-time generation of configurations is a feasible technique for use on highly adaptive embedded systems, where it may be used to provide precisely-tailored hardware support to tasks whose computational needs exceed the computational power of the CPU. The evaluation of the suitability of this approach for specific cases requires that all system aspects be considered. Although the time required for routing makes the approach unsuitable for applications requiring very fast generation of bitstreams, several classes of applications may be able to accommodate the delays involved and profit from the increased flexibility provided by this approach.

Acknowledgments

The authors would like to thank C. Ababei for providing some of the benchmarks used in section 4.

References

[1] J. Gause, P.Y.K. Cheung, and W. Luk. Reconfigurable computing for shape-adaptive video processing. *IEE Proceedings - Computers and Digital Techniques*, 151(5):313–320, 2004.

[2] K. Paulsson, M. Hiibner, J. Becker, J.-M. 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)*, pages 415–422, 2007.

[3] Miguel L. Silva and João C. Ferreira. Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems. *IET Computers & Digital Techniques*, 1(5):461–471, 2007.

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

[5] Ian Robertson and James Irvine. A design flow for partially reconfigurable hardware. *ACM Transactions on Embedded Computing Systems*, 3(2):257–283, 2004.

[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. International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 1–6, 2006.

[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*, pages 343–348, 2002.

[8] Y.E. 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)*, pages 1–4, 2006.

[9] Heiko Kalte and Mario Porrmann. REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 403–412. ACM, 2006.

[10] 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)*, pages 1–4, 2006.

[11] H. Tan and R. F. DeMara. A multilayer framework supporting autonomous run-time partial reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(5):504–516, 2008.

[12] Yaska Sankar and Jonathan Rose. Trading quality for compile time: ultra-fast placement for FPGAs. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 157–166. ACM, 1999.

[13] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *Proceedings of the 2001 ACM/SIGDA 9th International Symposium on Field-Programmable Gate Arrays*, pages 29–36. ACM, 2001.

[14] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proc. 41st Design Automation Conference*, pages 954–959, 2004.

[15] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[16] Jorge Suris, Cameron Patterson, and Peter Athanas. An efficient run-time router for connecting modules in FPGAS. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 125–130, 2008.

[17] Miguel L. Silva and João C. Ferreira. Generation of partial FPGA configurations at run-time. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 367–372, 2008.

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

[19] Cristinel Ababei and Kia Bazargan. Non-contiguous linear placement for reconfigurable fabrics. *International Journal of Embedded Systems*, 2(1/2):86–94, 2006.

[20] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenberg. A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pages 126–134. ACM, 2000.

[21] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *In International Symposium on Microarchitecture*, pages 330–335, 1997.