Journal of Systems Architecture 58 (2012) 24–37

Contents lists available at SciVerse ScienceDirect

Journal of Systems Architecture

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

Run-time generation of partial FPGA configurations

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 Porto, Faculdade de Engenharia, Universidade do Porto, R. Dr. Roberto Frias, 4200-465 Porto, Portugal

ARTICLE INFO

Article history:
Received 15 December 2010
Received in revised form 30 September 2011
Accepted 1 October 2011
Available online 8 October 2011

Keywords:
Reconfigurable computing
Run-time reconfiguration
Run-time bitstream generation
Adaptive embedded systems

ABSTRACT

This paper presents and evaluates a method of generating partial bitstreams at run-time for dynamic reconfiguration of sections of an FPGA. The method is intended for use in adaptive embedded systems that employ run-time reconfiguration to achieve high flexibility and performance. The proposed approach combines partial bitstreams of coarse-grained components to produce a new partial bitstream implementing a given circuit netlist. Topological sorting of the netlist is used to determine the initial positions of individual components, whose placement is then improved by simulated annealing. Connection routing is done by a breadth-first search of the reconfigurable area based on a simplified resource model of the reconfigurable fabric. The desired partial bitstream is constructed by merging together the default bitstream of the reconfigurable area, the relocated partial bitstreams of the components, and the configurations of the switch matrices used for routing. The approach is embodied in a code library that applications can use to create new bitstreams at run-time. For the members of a set of 29 benchmarks (both synthetic and application-derived) having between five and 41 components, the complete process of bitstream generation takes between 8 s and 35 s when running on an embedded PowerPC 405 microprocessor clocked at 300 MHz.

1. Introduction

This paper proposes and evaluates a method of generating 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 manage the loading of the newly created bitstreams to a specific FPGA area without disturbing the operation of other parts of the system. Highly adaptive embedded systems may employ the creation of partial bitstreams at run-time in situations where it is impractical to create them all 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. 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 the whole process must execute in a resource-limited embedded platform, simple algorithms are employed with the purpose of obtaining acceptable solutions in a reasonable time. Placement is based on sorting the components in topological order; the placement is then improved by simulated annealing. Routing is performed by finding the shortest path from a source terminal to the target terminal for successive nets; information of previous routes is used to speed-up the routing process. The complete run-time bitstream generation process has been implemented in the C programming language for execution by the PowerPC 405 embedded in Virtex-II Pro FPGAs. This implementation was used to evaluate the proposed approach with synthetic and application-derived benchmark circuits containing between five and 41 components: the whole process takes between 8 s and 35 s with the microprocessor running at 300 MHz.

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. The proposed approach to the generation of configurations imposes some requirements on the hardware infrastructure and assumes the existence of a design flow that produces all the data required for run-time operations: Section 3 describes the target hardware platform, gives a general description of the bitstream generation process, and describes the models used to represent components and FPGA routing resources. The placement of components is described in Section 4, while Section 5 explains the algorithm used for connecting the components. Results obtained by a proof-of-concept implementation for the target hardware platform are reported and discussed in Sections 7 and 8, respectively. Finally, Section 9 concludes the paper.

* 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.

1383-7621/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2011.10.001

2. Background and related work

Advanced embedded systems must meet challenging performance requirements under resource constraints. Reconfigurable computing, including run-time reconfiguration (RTR), is a natural approach to be exploited in this context [3], and is expected to play an important role in future heterogeneous multi-core embedded systems [4]. In the context of FPGA-based systems, run-time reconfiguration (RTR) designates the capability to alter the hardware design realized by the FPGA during the course of the execution of an application. The use of RTR promotes resource sharing [5], and increases time and power performance due to hardware specialization, even when considering the reconfiguration overhead [6,7]. Additionally, the combination of specialization and resource sharing may allow better power/cost trade-offs to be made [8].

Implementation and global performance of run-time reconfiguration are very dependent on the organization of the reconfigurable fabric and corresponding bitstream formats, and raise many issues related to system development and run-time resource management [9,10]. One of the issues concerns the generation of the required partial configurations. This is commonly done at design time, when all possibly useful partial configurations must be specified and created [11,12]. Several approaches to the relocation of partial bitstreams have been proposed, including both software tools [13,14] and hardware solutions [15,16]. Bitstream relocation is explicitly included in recent design flows [17].

In all cases mentioned 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 [18]. 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.

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 [19], which describes 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 studied in [20]: by combining different algorithms for placement and routing, the authors obtain 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 by [21]: it is able to produce good hardware circuits using 13 times less memory and executing 10 times faster than VPR [22] (running on a desktop computer).

A channel router for the Wires-on-Demand RTR framework is described in [23]. 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 less and execution is four orders of magnitude faster, for an average increase in delay of 15% (over a set of seven small benchmarks).

A different approach, run-time specialization of bitstreams, is described in [24]. Run-time parameterizable configurations from Boolean circuits are produced at design time, where some of the configuration bits are expressed as a function of a set of parameters. Given a specific parameter value, run-time specialization involves evaluating these functions in order to define the corresponding configuration bits. In this case, run-time specialization is equivalent to selection from a set of configurations defined at design time.

The bitstream assembly approach of [18] does not rely on the synthesis of logic descriptions, so it is a good candidate for implementation in an embedded system. A first, limited implementation is described in [25]: it is fast (less than 22 ms required for bitstream creation), but assumes a left-to-right data flow and performs routing by selecting routes from a predefined set. Only connections between adjacent components are allowed. That work is extended in [26] to include run-time routing, but limitations like unidirectional data flow and connections limited to adjacent components still apply. In both cases components are considered as black boxes, with all input terminals on the left side and all output terminals on the right.

In comparison, the implementation presented in this paper, while following the same general approach, puts more emphasis on flexibility and removes many restrictions of the previous works: it handles general netlists (not just acyclic netlists), imposes no constraints on the relative placement of the components nor on the positions of their terminals, and routes connections through the components. Furthermore, placement quality can be improved by simulated annealing, and routing can re-use information from previously established connections. The new approach is evaluated for a significantly larger number of benchmark circuits, and exhibits better execution times and improved results compared to [26].

3. Run-time bitstream generation

3.1. Hardware platform

The details and trade-offs of a working RTR implementations are closely tied to the internal organization of the reconfigurable fabric and to the format of the bitstream. In order to have a system for evaluation, it is necessary to take the specific device in consideration. Our proof-of-concept implementation targets the hardware platform used in [25,26]. Its top-level organization is shown in Fig. 1. It is built around a Virtex-II Pro FPGA, because this device supports active partial reconfiguration and includes an internal access port to the configuration memory.

The zone labeled "dynamic area" is reserved for RTR. The fixed base system is comprised by the other blocks. The configuration memory controller is responsible for handling the actual transfer of the bitstreams to the configuration memory through the internal configuration access port (ICAP). The dynamic area control unit supports the communication between the base system and one

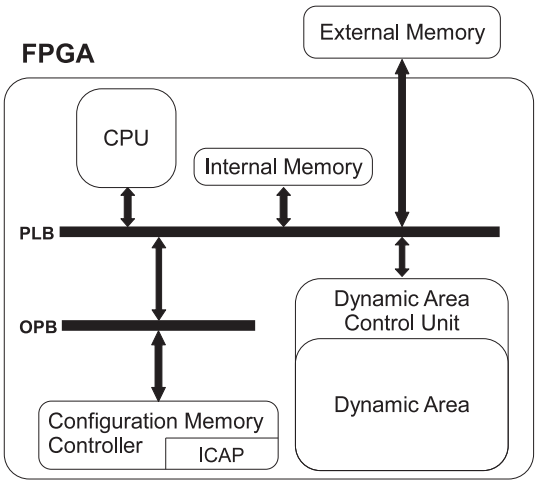

Fig. 1. Platform used for the proof-of-concept implementation of run-time generation of bitstreams. PLB: 64-bit Processor Local Bus; OPB: 32-bit On-chip Peripheral Bus; ICAP: Internal Configuration Access Port.

Table 1
Subset of FPGA routing resources that may be used for interconnecting components.

Type	# of resources	Comments
Direct	8×2	Connections to the 8 closest neighbors
Double lines	4×10	Short vertical or horizontal connections (1 or 2 CLBs away)
Hex lines	4×10	Longer vertical or horizontal connections (3 or 6 CLBs away)
CLB outputs	8	The eight unregistered CLB outputs
CLB inputs	32	Inputs to eight 4-input LUTs per CLB

dynamic area. The unit is connected to the processor's local bus in order to provide a fast communication path between application programs and the dynamically reconfigured modules. The external memory is used to store both the libraries of hardware components and application-specific data. More details about the hardware platform can be found in [18].

3.2. Resource models

Virtex-II Pro FPGAs have a segmented interconnection architecture, with segments connected by a regular array of switch matrices, which are connected between themselves and to other resources (like CLBs and BRAMs) [9,27]. 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. Long lines (i.e., bidirectional wires that distribute signals across the full device height or width) are excluded, because they can interfere with circuitry outside of the target dynamic area. It is unnecessary to consider other dedicated routing resources (like carry chains), because they have no bearing on the connections that are to be established at run-time. The resulting model of the switch matrix associated with each CLB contains 136 pins, distributed as shown in Table 1.

In our implementation the dynamic area is modeled as a two-dimensional array of switch matrices, and an internal data structure based on the simplified model just described is used to keep track of resource usage.

Components are rectangular areas of the reconfigurable fabric with fixed height and width (measured in number of CLBs) [18]. Each component employs the interface macros based on look-up tables (LUTs) described in [28], so that inputs and outputs must be located on the component's periphery. Components are not allowed to overlap, but interconnections may go through them, if free routing resources are available inside the component. The

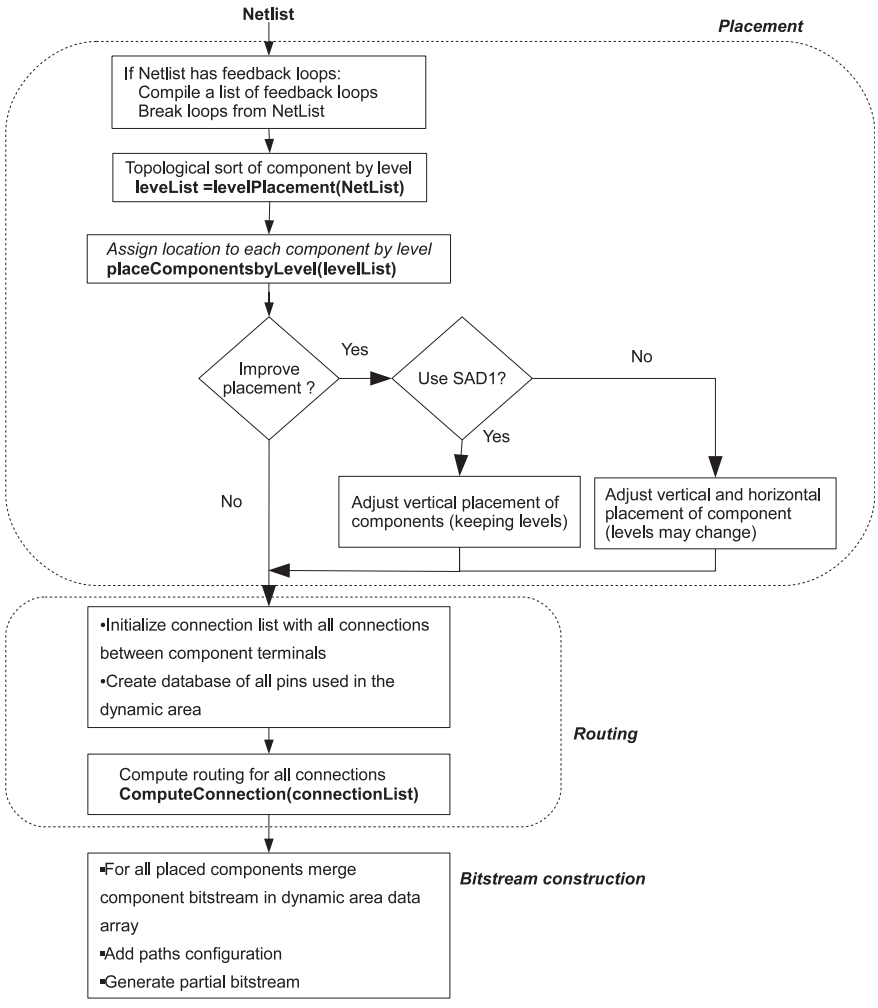

Fig. 2. Flow chart of bitstream generation process.

placement of components must obeys the restrictions imposed by the internal resource organization of the FPGA, as discussed in [18].

The standard synthesis flow produces a full bitstream for each component, which must be post-processed in order to extract the relevant partial bitstream and to add more information for use at run-time. The bitstream post-processor included in the extended version of BitLinker developed for this work employs the resource model of Table 1 to represent routing resource utilization in components. The routing resources used by the component are determined by examining the corresponding partial bitstream. Active CLB outputs are determined by checking the corresponding bits in the bitstream. For each such output, the post-processor traces back the path of the outgoing signal through the switches matrices, taking note of which pins are used. Only the subset of pins belonging to the model is examined, as these will be the only ones that might be used for connecting the components at run-time. Other resources, like carry chains, are not examined, since they will never be modified in that process.

3.3. Bitstream generation process

In this work, the creation of a new partial bitstream involves three major tasks: placement, routing, and bitstream construction (Fig. 2). The first task processes the netlist and selects locations for the components. The second task uses the routing algorithm of Section 5 to find the paths between component terminals. The final step converts the internal representation of the solution to the actual partial bitstream. The conversion is done in software by relocating the component bitstreams to the positions determined in the first step, and merging them with the default partial bitstream of the dynamic area. This process is performed as described in [18] for off-line bitstream generation. The partial bitstream is further modified to include the routes found in the second step.

All the tasks are performed by the embedded CPU available in the FPGA. The main subroutines are shown in bold in Fig. 2, and are described in the following sections.

4. Placing components

The placement phase takes a netlist specifying the components to be used and the unidirectional connections between their terminals. The general approach to placement is to find an arrangement of components in columns (a stripe) 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.

The first step of the algorithm is to group the components by levels, as described in Alg. 1. The first level contains the components with at least one input connected to the interface of the dynamic area. Second level components have all their input terminals connected to first-level components, third-level components have all inputs connected to previous levels, and so forth. If a component has more than one source, the component will be assigned to the level following the highest-numbered source. The level assignment is done by a depth-first search of the netlist. For netlists without cycles this is equivalent to processing the components in topological order. Cycles in the netlist are detected before level assignment by checking if the successors of the current component have already been visited (first box of Fig. 2). In this case, the level of the successor components is not changed.

The next step is to determine the columns that make up each stripe and to define the position of each component in the stripe. Two examples of component placement inside a stripe are displayed in Fig. 3. The position of a component in a stripe is affected by the resources it uses. In particular, components that internally use BRAMs must be assigned locations so that BRAMs are in the correct relative positions to the component's bounding box. The stripe width may have to be increased until an appropriate area of the fabric is included.

Procedure `PlaceComponentsByLevel` implements stripe-based placement as described Alg. 2 (with error handling omitted for clarity). Location assignment proceeds by processing each level in succession (line 2) and placing the components from top to bottom (line 5). If possible, a new component is placed just below the previous one and at the right edge of the stripe. However, finding appropriate locations for components with resources like BRAMs may require offsetting the component from the default location. As a result, it is possible that some components are not located right at the edges of the stripe.

The starting column assigned to a given 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. This is determined by the width of the components and by the compatibility of the component resources with the destination area (line 19). 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 (line 13).

Procedure `PlaceComponentsByLevel` creates an initial assignment of components to locations. The assignment can then be refined by simulated annealing (SA) [29]. Two versions of this procedure have been implemented (see Fig. 4). The simpler version (SA1D) tries to choose the best position for the components inside each stripe. The second version (SA2D) expands the first one by allowing a component to change stripes. Since SA2D is more

Algorithm 1: Function `LevelAssignment`(*netlist*)

Data: Netlist of all components
Result: Array of L of subsets of components

$i \leftarrow 0$
while ¬`Empty`(netlist) **do**
  $S \leftarrow$ subset of those nodes (from netlist) that have no incoming edges
  Remove from netlist all outgoing edges starting from nodes in S
  Remove S from netlist
  $L[i] \leftarrow S$, $i \leftarrow i + 1$
end
return L

Algorithm 2: Procedure `PlaceComponentsByLevel(levelList)`

```
   Input: Netlist levelList of all components, sorted by levels
 1 x ← 0                                              // first column of stripe
 2 foreach S ∈ levelList do                          // components of current level
 3     y ← 0                                                       // row index
 4     xLevel ← x                                   // leftmost column of stripe
 5     foreach comp ∈ S do                                  // ordered scan of S
 6         y1 ← y
 7         if HasBRAM(comp) then                    // component with internal BRAM
               // downward search for row with BRAMS
 8             while ¬MatchBRAMRows(comp, y1) do
 9                 y1 ← y1 + 1
10             end
11         end
12         x1 ← xLevel
           // rightward search for specific sequence of columns
           // that matches internal component organization
13         while ¬MatchColumns(comp, x1) do
14             x1 ← x1 + 1
15         end
16         AddToPlacementList(comp, x1, y1)                    // found place
17         // align search position for next component
18         y ← y1 + Height(comp)
19         if Width(comp) + x1 > x then
20             x ← x1 + Width(comp)
21         end
22     end
23 end
```

flexible than SA1D, the former is expected to provide better results than the latter, possibly at the cost of increased running time.

The objective of the optimization step is to minimize the distance between connected terminals in all stripes, so the cost associated with each placement is the sum of the Manhattan distances between net terminals. For connections to multiple inputs, the distance from input terminal to each output terminal is included.

At each temperature *T*, algorithm SA1D randomly generates 50 candidate solutions by exchanging components in random stripes.

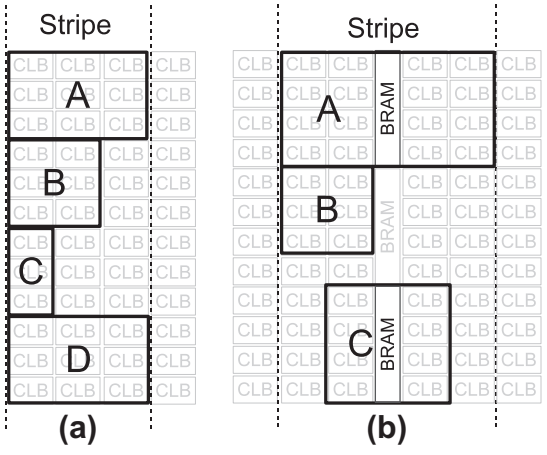

Fig. 3. Placing components in stripes. (a) Typical placement for components that only have CLBs; (b) placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

The cost difference Δ between a candidate solution and the current solution is determined. A candidate solution becomes the current solution, if $e^{-\Delta/T} > R$, with values of R produced by a uniform random generator ($0 \leqslant R \leqslant 1$). A better candidate always becomes the current solution; a worse candidate becomes the current solution with a probability that decreases with T.

The initial value of T is 10,000, and temperature is decremented at a cooling rate of 0.75. The algorithm stops after a predefined number of temperature decreases (currently, 50), or if it fails to make any improvement for a fixed number of successive temperature decreases (currently, 10). The algorithm keeps track of the placement with the lowest cost found so far (even if it is not the current solution), which becomes the final result when the iterative process ends.

Algorithm SA2D also starts from an initial placement based on the levels of the circuit graph. It is more flexible than SA1D because

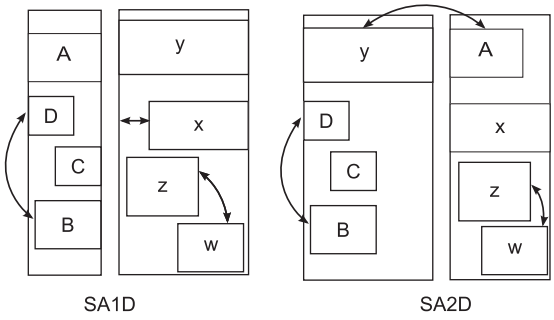

Fig. 4. Permissible changes of component location for the two implemented versions of simulated annealing.

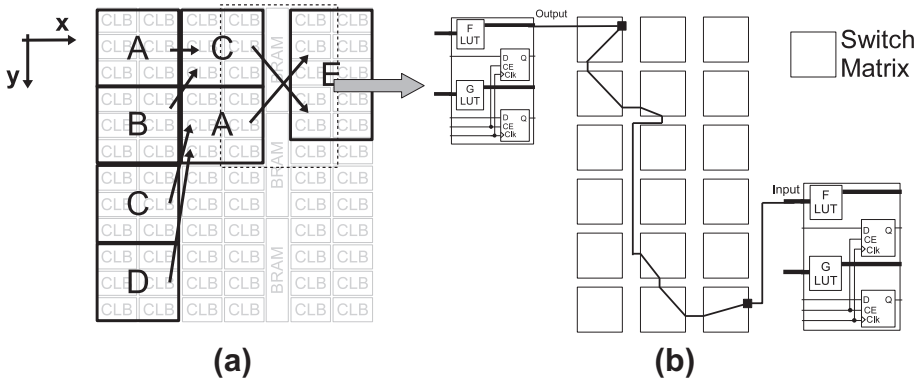

Fig. 5. Routing between stripes. (a) Placed components with indication of connections to be established; (b) detail of routing area (dashed box) showing one possible route connecting C to E.

it allows for components to be moved vertically (inside the same stripe) and horizontally (to any of the two adjacent stripes). Empty spaces in a stripe act as virtual components that can take part in the swaps. An empty stripe is assumed to follow the last physical

Algorithm 3: Function `LinkPins(source, sink)`

Data: Start pin source, end pin sink

```
1  examinedPins ← ∅, newCurrentPins ← ∅
2  currentPins ← source                    // pins for search expansion
3  distLimit ← GetDistance(source, sink), tmpDistLimit ← distLimit
4  while currentPins ≠ ∅ do
5      foreach p ∈ currentPins do
6          neighborhood ← list of pins connected to p
7          foreach pin ∈ neighborhood do
8              if pin = sink then                          // sink reached
9                  newPath ← RetracePathTo(source)
10                 AddPath(newPath) // add path to global solution
11                 return success
12             end
13             else if (InArea(pin) & pin ∉ usedPins &
               pin ∉ examinedPins &
               GetDistance(pin, sink) < distLimit + distTolerance) then
                   // add pin to search frontier
14                 examinedPins ← examinedPins + pin
15                 newCurrentPins ← newCurrentPins + pin
16                 if GetDistance(pin, sink) < tmpDistLimit then
17                     tmpDistLimit ← GetDistance(pin, sink)
18                 end
19             end
20         end
21     end
22     distLimit ← tmpDistLimit
23     currentPins ← newCurrentPins
24 end
25 return fail
```

stripe, in order to serve as a possible target for swaps. To limit the neighborhood search, the number of virtual components per stripe is restricted to three at most (limited by the maximum allowed stripe height). For components with multipliers or BRAMs, the "neighbor" stripes are the closest stripes with blocks of those types, not the ones that are adjacent.

Since routes can cross stripes by going through the components or by using empty spaces, a stripe density value is calculated, which is given by the percentage of occupied CLBs. For every route connecting terminals that are not in adjacent stripes, the average density of all stripes crossed by the route is calculated, and a 20% penalty is added to the route cost, resulting in the following cost function:

$$\text{New\_cost} = \text{Manhattan\_distance} \times (1 + 0.2 \times \text{Average\_density}).$$

SA2D does not impose a hard limit on the stripe height. Instead, it awards a 10-fold penalty increase to the cost of any net that has terminals outside the limits of the dynamic area. As a consequence overly tall stripes tend to break up by moving components to other stripes. The current SA2D implementation uses a cooling rate of 0.8, and may decrease temperature T up to 200 times. The final placement is only accepted if no penalty is associated with its cost; otherwise, the circuit is deemed too big for the target area, and placement fails.

5. Connecting components

In physical terms, component terminals are pins of the switch matrices. The component inputs correspond to pins connected to LUT inputs, while component outputs correspond to pins connected to CLB outputs (see last two rows of Table 1). 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 (see Fig. 5).

As shown in Fig. 2, the routing task starts by building a list of connections between component terminals (called `connectionList` in the algorithms). An N-sink net is converted to N individual connections. A database of used switch matrix pins (called `usedPins`) is built from the information available about each component. The database is updated to keep track of all pins that are used by the routed connections.

Procedure `LinkPins` (Alg. 3) is used to find a path from a source pin to one sink pin: it performs a breadth-first search for the shortest path between the source and sink terminals (line 4). The actual search area starts as the smallest rectangle that encloses all pins used as terminals of the connection to be established (line 3), and is reduced during the search, thereby limiting the number of segments that are considered (line 17). Restricting the search area in this way may cause some segments to be left out of consideration, but reduces the search effort. The search procedure measures path length as the number of its segments, which is, for FPGAs, a better metric than geometric distance [30].

During the search, the algorithm maintains a list of the pins that belong to the border of the expanding search (called `currentPins`). Initially, only the source of the net belongs to this set; during execution, the successors of each visited pin are added. Every pin added to this set includes a reference to its predecessor on the search path. As the search is extended to adjacent pins, variable `examinedPins` keeps track of which pins have been processed in order to avoid endless loops and ensure that only shortest paths are considered (line 14). When the sink is reached, a path to the source is determined by retracing through the sequence of interconnection segments (line 8) and recorded (line 10).

Algorithm 4: Procedure `ComputePaths(connectionList)`

```
   // build table of similar connections for route reuse
 1 similarConnections ← ComputeSimilarConnections(connectionList)
   // process each connection
 2 currentSource ← GetSource(connectionList[0])
 3 foreach conn ∈ connectionList do
 4 |   if currentSource ≠ GetSource(conn) then
   |   |   // all connections from current source processed
 5 |   |   usedPins ← usedPins + GetSolutionPaths(currentSource)
 6 |   |   currentSource ← conn
 7 |   end
 8 |   if GetSimilarConnection(conn) then
   |   |   // similar path exists:  adjust and check
 9 |   |   path ← AdjustPath(GetSimilarPath(conn))
10 |   |   if CheckPath(path) then
11 |   |   |   solutionPaths ← solutionPaths + path
12 |   |   |   continue with next connection
13 |   |   end
14 |   end
15 |   LinkPins(GetSource(conn), GetSink(conn));
16 end
```

The search area is limited by the test in line 13 of Alg. 3. An adjacent pin is only accepted for further processing if it is inside the dynamic area, is free and is within the bounds of the current search area (`distLimit`). In this last case, an empirically determined small tolerance is accepted (`distTolerance` = 2 in the implementation used). A pin may not be free because it was in use before the procedure started (registered in `usedPins`), or because it has been visited during the current search (registered in `examinedPins`). In addition, only routing resources that completely fit in the reserved area are used (code not shown in Alg. 3). For instance, hex lines that extend beyond the reserved area are not considered. This ensures that circuits outside the reserved area are not affected.

The `LinkPins` procedure is used by the main routing procedure shown in Alg. 4. This algorithm processes connections in sequence (line 3). In addition all connections that have the same source are processed consecutively; this is ensured by the way `connectionList` is built. Since connections with the same source may share pins, the global database `usedPins` is only updated in line 5.

In order to reduce execution time, the algorithm includes a preliminary step before each breadth-first search. In this step, the program checks whether a connection has already been made between terminals in the same relative position to each other (line 8). If this is the case, the program tries to reuse the previous route after shifting it to account for the positions of the current terminals. If all the required routing resources are free, the adjusted version is used for the current connection (line 10). In order to implement the route reuse heuristic, a list of "similar" connections (i.e., connections with end points in the same relative positions) is constructed at the start of the procedure (line 1).

6. Bitstream construction

For Virtex-II Pro devices, bitstream data is grouped in sections that correspond to columns of logic resources in the FPGA [31]. Each section is divided in frames, which correspond to a one-bit column of configuration data for the total height of the device. A frame is the smallest configurable segment of a Virtex-II Pro configuration, and configures many different logic resources along that vertical line. All the logic and routing information for a CLB and corresponding route matrix is split over the 22 frames that compose a CLB column. If we consider these 22 frames as an array of data, we can associate each CLB configuration with a contiguous group of data rows.

The largest amount of information in the internal database concerns the specification of how one interconnection matrix can be configured. The database specifies the output ports to which a given input port can be connected. For each connection, two coordinates specify the relative bitstream position that must be altered: The first specifies the corresponding frame within the 22 frames of a CLB column, the second gives the position of the bit in that frame. There is also the corresponding information for BRAM columns.

In addition, the database identifies the segments connected to each port by indicating the relative position of the switch matrix to which the other end of segment is connected, together with the corresponding port. For instance, it may specify that the segment connected to switch matrix input port number 1 is connected to output port 8 of a switch matrix located two CLB rows above. Port numbers are identified by indexes into an internal data structure. For matrix ports connected to CLB inputs or outputs, the corresponding CLB pin is specified.

Table 2
Basic structural characteristics of all benchmark circuits used in the evaluation of the proof-of-concept implementation.

	Circuit	# In	# Out	# Components	# Nets	# Levels
P1	Pipeline 1	24	24	8	120	4
P2	Pipeline 2	32	32	9	160	4
D1	DAG 1	16	8	5	72	3
D2	DAG 2	32	32	7	112	3
D3	DAG 3	32	32	12	208	4
D4	DAG 4	32	32	12	264	5
D5	DAG 5	32	32	12	256	5
C1	Cycle 1	40	32	8	96	4
C2	Cycle 2	48	32	12	160	4
C3	Cycle 3	48	32	16	224	5
Benchmarks adapted from [33]						
T1	tg01	40	8	9	72	4
T2	tg02	80	8	19	152	5
T3	tg03	48	8	11	88	4
T4	tg04	80	8	19	152	6
T5	tg05	80	8	19	152	5
T6	tg06	120	8	25	200	5
H1	Honeywell-intfc01	24	40	13	104	4
H2	Dft	56	32	12	64	3
H3	Honeywell-versatil01	24	40	20	144	5
H4	Honeywell-intfc02	48	56	21	152	7
H5	Honeywell-fft01	40	64	23	168	6
H6	Honeywell-fft02	56	56	24	184	7
H7	MediaBench-jpeg	56	128	27	200	5
H8	Honeywell-fft03	48	32	28	240	8
H9	Honeywell-versatil02	72	104	41	328	8
Benchmarks adapted from [34]						
M1	HAL	40	24	11	64	4
M2	MPEG Motion Vector	104	24	32	232	6
M3	MESA Horner Bezier	40	16	18	128	8
M4	Auto Regression Filter	48	16	28	288	8

Table 3
Time required for placement (P), routing (R), and partial bitstream construction (C) for the benchmark circuits (in seconds) on the 300 MHz embedded PowerPC.

	No SA				SA1D				SA2D			
	P	R	C	Total	P	R	C	Total	P	R	C	Total
P1	0.03	15.10	0.10	15.23	0.07	15.05	0.10	15.22	0.12	15.02	0.10	15.15
P2	0.04	19.48	0.15	19.67	0.07	19.38	0.15	19.60	0.15	18.92	0.15	19.11
D1	0.02	9.07	0.04	9.13	0.04	8.93	0.04	9.01	0.09	8.99	0.04	9.23
D2	0.03	13.48	0.08	13.59	0.05	13.35	0.08	13.48	0.10	12.81	0.08	13.01
D3	0.04	25.77	0.26	26.07	0.09	23.96	0.25	24.30	0.21	21.25	0.25	21.61
D4	0.05	32.25	0.33	32.63	0.10	30.99	0.32	31.41	0.19	28.21	0.33	28.42
D5	0.05	31.31	0.31	31.67	0.10	30.21	0.31	30.62	0.18	28.28	0.31	28.50
C1	0.03	12.10	0.08	12.21	0.07	11.55	0.08	11.70	0.12	11.01	0.08	11.36
C2	0.05	19.29	0.20	19.54	0.09	18.95	0.20	19.24	0.19	17.81	0.20	18.45
C3	0.06	27.40	0.37	27.83	0.13	25.47	0.38	25.98	0.25	23.73	0.37	24.68
T1	0.04	8.65	0.07	8.76	0.07	8.61	0.06	8.74	0.16	8.64	0.07	8.84
T2	0.07	18.57	0.30	18.94	0.14	18.41	0.29	18.84	0.28	18.57	0.30	18.93
T3	0.05	10.64	0.10	10.79	0.08	10.49	0.10	10.67	0.17	9.81	0.10	10.23
T4	0.08	18.95	0.29	19.32	0.14	16.47	0.29	16.90	0.31	14.96	0.29	15.60
T5	0.07	18.26	0.30	18.63	0.16	17.70	0.29	18.15	0.29	16.08	0.30	16.68
T6	0.10	24.63	0.51	25.24	0.18	23.43	0.52	24.13	0.38	22.89	0.52	23.35
H1	0.05	13.03	0.14	13.22	0.09	12.89	0.14	13.12	0.22	11.71	0.14	12.13
H2	0.05	8.08	0.08	8.21	0.09	7.77	0.08	7.94	0.18	7.41	0.08	7.96
H3	0.08	17.39	0.30	17.77	0.15	16.88	0.29	17.32	0.34	15.56	0.29	15.97
H4	0.09	18.62	0.33	19.04	0.18	17.95	0.32	18.45	0.33	17.72	0.34	18.35
H5	0.10	20.68	0.40	21.18	0.19	19.05	0.39	19.63	0.40	18.43	0.39	18.93
H6	0.09	22.65	0.46	23.20	0.21	21.73	0.44	22.38	0.40	20.88	0.46	21.57
H7	0.11	24.99	0.55	25.65	0.22	24.85	0.55	25.62	0.43	23.08	0.55	23.81
H8	0.11	29.13	0.70	29.94	0.21	28.04	0.71	28.96	0.42	26.21	0.69	27.15
H9	0.16	39.97	1.41	41.54	0.35	36.31	1.38	38.04	0.62	34.24	1.35	34.95
M1	0.04	8.04	0.07	8.15	0.09	8.04	0.07	8.20	0.16	8.02	0.07	8.34
M2	0.13	29.04	0.75	29.92	0.27	25.94	0.76	26.97	0.47	23.64	0.76	24.24
M3	0.08	15.47	0.24	15.79	0.14	14.95	0.24	15.33	0.31	14.03	0.24	14.67
M4	0.10	35.95	0.83	36.88	0.24	32.38	0.82	33.44	0.47	30.58	0.84	31.19

Table 4
Characteristics of the run-time-generated implementations of the benchmark circuits. AS: average number of segments in connections; MS: maximum number of segments; ΔAS: reduction in the average number of segments. The reconfiguration time given in the last column is for the bitstream produced with SA2D. The symbol = marks entries that remain unchanged when simulated annealing is employed.

	No SA			SA2D			ΔAS (%)	Reconf. time (ms)
	BB	AS	MS	BB	AS	MS		
P1	12 × 9	4.19	6	=	4.17	=	0.3	3.82
P2	12 × 12	4.02	6	=	3.93	=	2.2	3.83
D1	8 × 6	4.45	6	=	4.38	=	1.6	2.46
D2	9 × 16	6.85	10	9 × 14	6.56	9	4.2	2.76
D3	12 × 24	6.1	9	15 × 18	5.05	8	17.2	4.65
D4	16 × 32	7.48	11	16 × 24	6.53	10	12.6	4.92
D5	21 × 32	8.52	12	21 × 24	7.73	11	9.3	6.66
C1	12 × 20	8.52	12	12 × 18	7.78	11	8.8	3.8
C2	16 × 20	8.11	11	16 × 18	7.55	=	6.9	5.08
C3	16 × 24	8.49	12	16 × 20	7.38	11	13.1	4.87
T1	10 × 11	4.19	6	=	4.15	=	1.0	3.06
T2	13 × 17	5.54	8	13 × 15	5.49	=	0.9	4.01
T3	10 × 18	5.26	7	10 × 16	4.83	=	8.2	3.11
T4	16 × 17	5.57	8	16 × 15	4.42	6	20.7	4.89
T5	13 × 17	5.87	8	13 × 15	5.13	=	12.6	3.95
T6	13 × 27	6.94	10	13 × 24	6.39	9	7.9	3.96
H1	21 × 16	5.82	8	=	5.18	7	11.0	6.42
H2	9 × 28	5.7	8	9 × 25	5.22	=	8.4	2.84
H3	18 × 29	6.68	9	18 × 26	5.98	8	10.5	5.58
H4	21 × 29	7.89	11	21 × 25	7.5	=	5.0	6.57
H5	18 × 33	7.23	11	18 × 28	6.49	10	10.3	5.61
H6	21 × 32	8.79	12	21 × 27	8.12	11	7.7	6.51
H7	15 × 32	7.22	10	15 × 28	6.65	=	7.9	4.6
H8	24 × 35	8.34	12	24 × 28	7.43	11	10.9	7.42
H9	24 × 56	8.34	12	24 × 40	7.09	11	15.0	7.51
M1	12 × 12	4.89	7	12 × 12	4.87	=	0.5	3.71
M2	18 × 24	5.67	8	18 × 20	4.61	7	18.8	5.58
M3	24 × 12	6.55	9	=	5.93	8	9.5	7.55
M4	24 × 24	6.33	9	24 × 20	5.41	8	14.5	7.55

Table 5
Running time improvement obtained by applying the route reuse heuristic after placement optimization with SA2D. Column "Impr." shows the running time improvement achieved by the heuristic.

Circuit	# Nets	Route reuse (%)	Time w/o heuristic (s)	Time w/heuristic (s)	Impr. (%)
H1	104	9.62	12.15	11.71	3.6
H2	64	12.50	8.31	7.41	10.8
H3	144	11.11	16.52	15.56	5.8
H4	152	15.79	18.92	17.72	6.3
H5	168	14.88	20.01	18.43	7.9
H6	184	10.87	22.81	20.88	8.5
H7	200	11.50	24.95	23.08	7.5
H8	240	9.17	27.33	26.21	4.1
H9	328	8.54	35.96	34.24	4.8
M1	64	12.50	9.02	8.02	11.0
M2	232	8.62	26.11	23.64	9.5
M3	128	12.50	15.32	14.03	8.4
M4	288	8.33	30.93	30.58	1.1

Bitstream construction is divided in three steps (see Fig. 2). In the first step, the component configuration information in written to the correct locations of the array that represents the dynamic area. In the second step, the configuration for the switch matrices is added to the array.

At the beginning of the first step, the bitstream of the empty dynamic area is used to initialize the working array of configuration information. For each component, the component library includes an array that contains the configuration data for all its logic resources. This information is extracted by BitLinker when the component library is created [18]. Since the placement phase already determined the position of each component, it is only necessary to copy all data from the component array to the corresponding positions of the dynamic area data array. Note that the placement step ensures that all constraints related to the positioning of the components (as detailed in [18]) are satisfied.

The second step takes a list of switch matrix connections that need to be configured, and uses the information contained in an internal database to determine the corresponding position in the dynamic area data array and what value must be set.

In the third step, the information in the data array is used to create a partial bitstream for all CLB columns of the dynamic area, that are actually used by the circuit. The format of Virtex-II Pro bitstreams is similar to the ones used by the Virtex and Virtex-II family [9,31], with some modifications to account for the presence of additional dedicated blocks (like the multipliers and the PowerPC cores). This step includes adding the appropriate header information, and formatting that data according to the rules of the bitstream format. This partial bitstream is used to configure the device through the ICAP controller.

7. Experimental results

The experimental evaluation of the proposed approach was performed by applying the algorithms from the previous sections to 29 benchmark circuits. The evaluation was done on a XUP Virtex-II Pro development system, which has a Xilinx XC2VP30-7 FPGA ([32]) and 512 MB of external DDR memory (PC-3200). The external memory contains the program code and all data, including the library of components. The evaluation used only one of the two embedded PowerPC 405 processors. The CPU operates at 300 MHz, and the 64-bit processor local bus connected to the memory controller has a 100 MHz bus clock.

Table 2 summarizes the structural characteristics of the individual benchmark circuits: the number of input and output ports, the number of components and nets, and the number of levels of the netlist (with cycles broken as discussed in Section 4).

The first group of benchmark circuits comprises three types of synthetic circuits. Netlists P1 and P2 describe pipelines made of components with 8-bit or 16-bit ports, with three and four stages. These benchmarks are used to verify that the algorithms process linear arrangements of components in the expected manner. Netlists D1–D5 are directed acyclic graphs (DAGs) with multiple components having 8-bit, 16-bit or 32-bit ports. Netlists C1–C3 include feedback cycles. The main purpose of this set is to represent netlists with cycles and long feedback connections.

The second group of benchmarks is based on data flow graphs used in [33]. Benchmarks T1–T6 consist of circuits that implement arithmetic expressions on 8-bit data. They evaluate how the algorithms perform for trees of components. Each expression is mapped to a binary tree, whose leaf nodes are the expression variables and whose root node represents the result of the expression. All internal nodes correspond to binary operations.

The third set of circuits comes from embedded system benchmarks. Circuits H1–H9 are based on data flow graphs adapted by [33] from the Honeywell [35] and MediaBench benchmarks [36]. All nodes process 8-bit data items. They are used to evaluate the application of the proposed approach to kernels of real embedded applications. Benchmarks M1–M4 come from additional data flow graphs taken from MediaBench [34] in order to produce further evidence of the applicability of the algorithms to real-world circuits.

The running times taken by the complete bitstream generation procedure on the proof-of-concept system are shown in Table 3. Results for the three possible placement options are shown. In addition to the total running time, the table gives the time required for placement, routing, and bitstream construction. The times shown correspond to the median value of five runs of each benchmark.

The main characteristics of the circuits generated at run-time are summarized in Table 4. For each circuit, Table 4 gives the final bounding box in CLBs (number of columns $\times$ number of rows), together with the average and maximum length of the interconnections (in number of segments). The reconfiguration time for the resulting bitstream is also shown.

Table 3 shows that using the SA1D improvement step almost always leads to longer running times when compared to using SA2D. Since SA2D is more flexible and therefore able to find better placement solutions, only results for the latter are included in Table 4.

The correctness of the generated bitstreams was verified by loading them to the FPGA and testing them with randomly-generated data. The results were compared to those obtained by straight implementations of the underlying data flow graphs in C. This procedure was followed for all benchmarks, except H5, H8, and H9, since they did not fit the dynamic area available in the implementation platform (23×32); for these cases, a manual check of the bitstream was done.

All versions used for evaluation apply the route reuse heuristic described in Section 5. The effect of using this heuristic was evaluated for the subset of application-derived benchmarks. Table 5 shows the percentage of connections that were routed using the heuristic, and the running times for two versions of the routing procedure (with and without re-use).

8. Discussion

8.1. Running time

Table 3 shows that in all cases placement is very fast, even when simulated annealing is used. The longest placement time (0.62 s) occurs for benchmark H9 and takes up 1.8% of the total execution time. Bitstream construction is also relatively fast: the worst case (H9) takes 1.41 s, which is 3.4% of the total time. Routing is, as expected, by far the most time-consuming step.

The use of SA1D improves most total running times very slightly: the largest reduction was by 3.5 s (8.4%) for benchmark H9. For benchmark M1, the use of SA1D increased total running time by 0.05 s. In all other cases, the increase in placement time was compensated by a reduction in routing time.

The use of SA2D generally improves on the total running time when compared to the other two options. The largest relative improvement occurs for T4: 3.72 s (19.3%). Again, benchmark M1 is the only one whose running time is degraded by SA2D: it takes 0.19 s longer than the base version (2.3%). In almost all cases where SA2D improves the running time, it also achieves better results than SA1D. Although there are some exceptions, the data show a clear trend of running time improvement with the use of better placement procedures, producing larger improvements for the more complex benchmarks.

Table 5 shows the impact of the route reuse heuristic. The average reuse is 11.2%, with individual values ranging from 8.3% to 15.5%. The average improvement in running time is 6.9% (minimum: 1.1%; maximum: 11.0%). The results show that this heuristic improves the global running time, but the relation between route reuse and running time is not direct. Maximum reuse (benchmark H4) and maximum time improvement (benchmark M1) occur for different circuits. In fact, benchmark H4 exhibits a below-average improvement in running time. In general, the route reuse heuristic is very cost effective, because it provides an improvement in running time at a very low implementation cost.

8.2. Quality of the generated circuits

Concerning the quality of the results in terms of area (bounding box) and length of the connections, Table 4 shows that, as expected, circuits with larger bounding boxes have higher average number of segments per connection and higher maximum interconnect lengths. The use of SA2D improves the bounding box of 23 benchmarks (79%). The pipeline benchmarks are one significant exception to the general improvements. This is to be expected, since the structure of these circuits is well matched to the basic placement procedure. Since the unoptimized procedure already produces a good placement, there is little room for improvement. Other exceptions (T1 and D1) are among the circuits of their type with the smallest number of levels. For this reason the additional flexibility of SA2D has no impact on the final result. The bounding box reductions are more significant for larger circuits. The largest improvement is exhibited by the H9, whose bounding box decreases from 24×56 to 24×40.

As regards connection length, Table 4 shows that the SA2D improvement step achieves a mean length reduction of 8.4%, with a minimum of 0.3% for circuit P1, and a maximum of 20.7% for circuit T4. The maximum connection length is reduced by one segment in 16 cases, and by two in one case (T4). For all other benchmarks, the maximum connection length remains unchanged by SA2D. The average length of the connections is also systematically improved, in the best cases by more than one segment: M2 shows a reduction of 18.7% from 5.67 to 4.61 for the average length.

We have also measured the bounding box and the average number of segments achieved by using SA1D (not shown in the tables). They are always equal or worse than the results obtained by SA2D. It can be concluded that the use of SA2D consistently improves both the quality of the results and the running time, when compared to the other two alternatives. The longer (but still relatively small) placement times are compensated by shorter routing times. SA1D, although it achieves a shorter placement time than SA2D, does not represent a favorable trade-off between overall speed and quality.

Reconfiguration time is proportional to the number of CLB columns used in the circuit. For the benchmarks used in this work it varies between 2.46 ms for D1 and 7.55 ms for M3 and M4 (Table 4). The values come from actual measurements performed with the implementation platform, so small variations (on the order of hundreds of millisecond) are not significant. The reconfiguration time for benchmarks H5, H8, and H9 was also measured by loading them into the FPGA, although the functionality could not be checked, since they did not fit in the dynamic area. For a given bitstream, the reconfiguration time is limited by the throughput of the communication channel to the ICAP and by the ICAP itself. The use of the techniques presented in [37] would improve this aspect.

8.3. Placement and routing algorithms

The placement and routing algorithms implemented for this work impose no restrictions on the locations of the components, nor on the position of the terminals. Compared to the approach of [26], the current implementation does not require acyclic netlists, and is able to route connections through the components. Routing is not restricted to components in adjacent stripes: the current implementation is able to connect terminals anywhere inside the dynamic area. This fact reduces the impact of placement choices on routing. The use of simulated annealing improves component placement, yielding circuits with smaller areas.

In addition to achieving a smaller area, improving placement also improves routing time and quality. The data of Table 3 show that the most significant improvement in routing time occurs for benchmark H8: with SA2D the routing time decreases 19.3%, for a global improvement of 15.9%. Using SA1D achieves smaller routing time improvements (9.1% for H8 and 10.7% for M2). For the benchmarks of Table 3, a better placement never degrades the routing time. However, in some cases, the improvement is not large enough to compensate for the additional placement time. This behavior typically occurs for the smaller benchmarks. For instance, benchmark D1 shows a slightly smaller routing time when using SA2D, but a larger total running

time due to the larger placement time. Benchmark T1 displays a similar behavior.

From the perspective of the impact upon routing time, the SA2D optimization step almost always performs better than SA1D. Benchmarks D1, T1 and T2 are the only ones where SA1D leads to a shorter routing time, although by a small margin of at most 0.9% (for T2). This behavior is in agreement with the global results discussed in Section 8.1, since routing time makes up for most of the global running time.

Although the benchmarks do not exhibit an uniform behavior, there is a clear trend indicating that more sophisticated placement procedures lead to shorter routing times, with larger improvement occurring for more complex benchmarks. In addition, as discussed in Section 8.2, the quality of the routes also improves when using SA2D (cf. Table 4). In all cases, using an more powerful version of the placement routine enables the routing procedure to achieve a reduction in the average number of segments per connection; in the majority of cases, the number of segments of the largest connection is also reduced. This is an expected result due to the use of the interconnection length as part of the cost function that the SA2D tries to minimize.

One of the main restrictions imposed on the implementation is the limited computational power available on the target systems, which leads to some compromises. The approach for routing described here does not ensure that a global optimum for all interconnections is obtained, since each net is treated in isolation, without considering its impact on subsequent nets. In addition, the dynamic restriction of the search area 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 negative impact of these design choices is limited by the use of coarse-grained components and the regular placement of the components, which naturally restrict the choices available for routing. The placement procedures matches the regular data flow of the computational kernels of many DSP-like applications, which are the probable targets for performance enhancement through dedicated hardware support. As shown by the benchmark circuits of Section 7, a large variety of circuits can be successfully routed under these constraints.

Using the fastest version, the complete process of bitstream generation takes between 8 s and 35 s. These running times make the current version unsuitable for applications that require a very fast turnaround time, like just-in-time compilation. There are, however, 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 changes) or that may operate temporarily with reduced quality.

Another application scenario 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 for 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 adding appropriate "glue" components to the design.

8.4. Other FPGA architectures

The software developed for this work is targeted at the Virtex-II Pro architecture, but a similar approach would be applicable to more recent generations (Virtex-4, 5 and 6). All these families maintain a similar heterogeneous architecture, with embedded blocks in a sea of CLBs connected by a segmented interconnection network. Some changes have relatively little impact. For instance, the existence of different types of embedded blocks would have little effect. Also, the fact that more recent families have 6-input LUTs (instead of 4-input LUTs) does not significantly affect the proposed approach, because no use is made of the internal organization of the logic cells.

Changes to the routing architecture have a more direct effect. For instance, the Virtex-5 family introduces a new diagonally symmetrical routing architecture with L-shaped lines replacing the straight hex lines. This change would affect the resource model described in Section 3.2, but not in a fundamental way.

The approach used in this work is naturally impacted by any change of the bitstream format. More recent families, organize the bitstream information along the same general lines, but with a key difference in that frames no longer span the total height of the device; instead frames have a standard height corresponding to a clock region [9]. This change makes it easier to reconfigure smaller areas of the device without affecting others. In addition, a finer frame granularity may contribute to shorter reconfiguration times. This type of change mostly impacts the bitstream construction step.

Any bitstream format change implies that components have to be re-synthesized, in order to have compatible configuration information. Assuming that enough information about the bitstream format is available to enable the manipulation of the switch matrix configuration, the same algorithms for placement and routing could be employed together with a revised bitstream construction procedure. We estimate that this step of the process is the one the would require more work when targeting more recent FPGA families.

9. Conclusion

This paper presents the implementation and evaluation of an embedded system that is able to generate partial bitstreams at run-time for the run-time reconfiguration of sections of a Virtex-II Pro platform FPGA. The goal is to obtain useful solutions in a short time. The system uses topological sorting to determine an initial position for individual coarse-grained components. The initial placement may be improved by a simulated annealing procedure. Two variants of the simulated annealing have been implemented: the more flexible one provides the best results, regarding both the quality of the final results and the total running times.

A procedure based on a breadth-first search over restricted areas determines the routes for the interconnections. For connections whose end points have relative positions identical to previously processed connections, the implementation first tries to use a shifted version of the original route. The overall computational effort is kept bounded by the use of a simplified resource model, fast placement procedure, the restriction of

routing to limited areas, and the use of a simple route reuse heuristic.

The results for a set of 29 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, but is under 35 s for all benchmarks (average: 18 s).

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 for 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 thank C. Ababei for providing some of the benchmarks used in Section 7. 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 Proc. Comput. Digit. Tech. 151 (2004) 313–320.
[2] K. Paulsson, M. Hbner, J. Becker, J. Philippe, C. Gamrat, On-line routing of reconfigurable functions for future self-adaptive systems – investigations within the ÆTHER project, in: Int. Conf. Field-Programmable Logic Appl. 2007 (FPL), pp. 415–422.
[3] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M.D. Ciano, J. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V.L. Barba, P. Cuvelier, B. Rousseau, P. Gelineau, An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications, EURASIP J. Embedded Syst. (2008).
[4] K.D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenström, O. Temam, High-Performance embedded architecture and compilation roadmap, in: Trans. High-Performance Embedded Arch. Compilers I, volume 4050 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 5–29.
[5] M.J. Wirthlin, B.L. Hutchings, Improving functional density using run-time circuit reconfiguration, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 6 (1998) 247–256.
[6] O. Prez, Y. Berviller, C. Tanougast, S. Weber, The use of runtime reconfiguration on FPGA circuits to increase the performance of the AES algorithm implementation, J. Universal Comput. Sci. 13 (2007) 349–362.
[7] J. Islam, P.W. Chun, W.J. MacLean, L. Kirischian, Lowering power consumption using run-time reconfiguration for stereo rectification, in: Proc. 2008 Canadian Conf. Electrical Comput. Eng. CCECE, IEEE, 2008, pp. 1693–1698.
[8] J. Becker, M. Hübner, M. Ullmann, Run-time FPGA reconfiguration for power-/cost-optimized real-time systems, in: VLSI-SOC: From Systems to Chips, volume 200 of IFIP Int. Federation Inform. Proc., Springer, 2006, pp. 119–132.
[9] P. Hsiung, M. Santambrogio, C. Huang, Reconfigurable System Design and Verification, CRC Press, 2009.
[10] M. Platzner, J. Teich, N. Wehn (Eds.), Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications, Springer, 2010.
[11] I. Robertson, J. Irvine, A design flow for partially reconfigurable hardware, ACM Trans. Embedded Comput. Syst. 3 (2004) 257–283.
[12] 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: Proc. Int. Conf. Field Programmable Logic Appl. (FPL), pp. 1–6.
[13] E.L. Horta, J.W. Lockwood, D.E. Taylor, D. Parlour, Dynamic hardware plugins in an FPGA with partial run-time reconfiguration, in: Proc. 39th Design Automat. Conf., 2002, pp. 343–348.
[14] Y. Krasteva, E. de la Torre, T. Riesgo, D. Joly, Virtex II FPGA bitstream manipulation: application to reconfiguration control systems, in: Proc. Int. Conf. Field Programmable Logic Appl., 2006 (FPL), pp. 1–4.
[15] H. Kalte, M. Porrmann, REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs, in: Proc. Third Conf. Comput. Frontiers, ACM, 2006, pp. 403–412.
[16] F. Ferrandi, M. Morandi, M. Novati, M.D. Santambrogio, D. Sciuto, Dynamic reconfiguration: core relocation via partial bitstreams filtering with minimal overhead, in: Proc. Int. Symp. System-on-Chip, 2006, pp. 1–4.
[17] H. Tan, R.F. DeMara, A multilayer framework supporting autonomous run-time partial reconfiguration, IEEE Trans. Very Large Scale Integrat. (VLSI) Syst. 16 (2008) 504–516.
[18] M.L. Silva, J.C. Ferreira, Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems, IET Comput. Digit. Tech. 1 (2007) 461–471.
[19] Y. Sankar, J. Rose, Trading quality for compile time: ultra-fast placement for FPGAs, in: Proc. Seventh ACM/SIGDA Int. Symp. Field-Prog. Gate Arrays, ACM, 1999, pp. 157–166.
[20] C. Mulpuri, S. Hauck, Runtime and quality tradeoffs in FPGA placement and routing, in: Proc. ACM/SIGDA Ninth Int. Symp. Field-Progr. Gate Arrays, ACM, 2001, pp. 29–36.
[21] R. Lysecky, F. Vahid, S.X.-D. Tan, Dynamic FPGA routing for just-in-time FPGA compilation, in: Proc. 41st Design Autom. Conf., 2004, pp. 954–959.
[22] V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.
[23] J. Suris, C. Patterson, P. Athanas, An efficient run-time router for connecting modules in FPGAS, in: Proc. Int. Conf. Field Programmable Logic Appl., 2008, pp. 125–130.
[24] K. Bruneel, D. Stroobandt, Automatic generation of run-time parameterizable configurations, in: Proc. Int. Conf. Field Programmable Logic Appl., 2008, pp. 361–366.
[25] M.L. Silva, J.C. Ferreira, Generation of partial FPGA configurations at run-time, in: Proc. Int. Conf. Field Programmable Logic Appl., 2008 (FPL), pp. 367–372.
[26] M.L. Silva, J.C. Ferreira, Creation of partial FPGA configurations at run-time, in: Proc. Euromicro Symp. Digit. Syst. Design, IEEE Computer Society, 2010, pp. 80–87.
[27] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, Xilinx, 2007, Version 4.7.
[28] M. Hbner, T. Becker, J. Becker, Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration, in: Proc. 17th Symp. Integr. Circuits Syst. Design, 2004, pp. 28–32.
[29] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (1983) 671–680.
[30] T. Taghavi, S. Ghiasi, A. Ranjan, S. Raje, M. Sarrafzadeh, Innovate or perish: FPGA physical design, in: Proc. 2004 Int. Symp. Phys. Design, ACM, 2004, pp. 148–155.
[31] Xilinx, Virtex Series Configuration Architecture User Guide (XAPP 151), 2004.
[32] Xilinx, Virtex-II Platform FPGA User Guide, Xilinx, 2007, Version 2.2.
[33] C. Ababei, K. Bazargan, Non-contiguous linear placement for reconfigurable fabrics, Int. J. Embedded Syst. 2 (2006) 86–94.
[34] G. Wang, W. Gong, B. DeRenzi, R. Kastner, Ant colony optimizations for resource- and timing-constrained operation scheduling, IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. 26 (2007) 1010–1029.
[35] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, H. Spaanenberg, A benchmark suite for evaluating configurable computing systems – status, reflections, and future directions, in: Proc. Eighth Int. Symp. Field Programmable Gate Arrays, 2000, pp. 126–134.
[36] C. Lee, M. Potkonjak, W.H. Mangione-Smith, MediaBench: A tool for evaluating and synthesizing multimedia and communications systems, in: Proc. 13th Annual IEEE/ACM Int. Symp. Microarchitecture, IEEE/ACM, 1997, pp. 330–335.
[37] C. Claus, R. Ahmed, F. Altenried, W. Stechele, Towards rapid dynamic partial reconfiguration in video-based driver assistance systems, in: P. Sirisuk, F. Morgan, T. El-Ghazawi, H. Amano (Eds.), Reconfigurable Computing: Architectures, Tools and Applications, Springer, 2010, pp. 55–67.

Miguel L. Silva received his B.S. in Electrical and Computer Engineering from the Faculty of Engineering of the University of Porto and a M.S. in Artificial Intelligence and Computation from the same University. He is currently a Ph.D. 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.