A runtime relocation based workflow for self dynamic reconfigurable systems design

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1109/FPL.2009.5272545">http://dx.doi.org/10.1109/FPL.2009.5272545</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>Version</td>
<td>Final published version</td>
</tr>
<tr>
<td>Citable link</td>
<td><a href="http://hdl.handle.net/1721.1/60253">http://hdl.handle.net/1721.1/60253</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Article is made available in accordance with the publisher’s policy and may be subject to US copyright law. Please refer to the publisher’s site for terms of use.</td>
</tr>
</tbody>
</table>
ABSTRACT

A self, partial and dynamic approach to reconfiguration makes it possible to obtain higher flexibility and better performance with respect to simpler approaches; however, the price for this improvement lies in the increased difficulties in the reconfigurable system creation and management, which become significantly more complex. An automated or semi-automated way to support this kind of systems would simplify the problem by raising the level of abstraction at which the designer has to operate. The aim of this work is the creation of a complete workflow to help the designer in the creation and management of self partially and dynamically reconfigurable systems: the designer should only specify the application, the reconfigurable device, the reconfiguration model (1D vs 2D) and the type of communication infrastructure, and the automated flow will deal with the subsequent steps down to the final architecture implementation. Among other aspects, the provided support includes the definition of area constraint for cores, the creation of an efficient runtime solution for core allocation management and the generation of a solution to obtain internal and fast relocation of cores.

1. INTRODUCTION

The proposed work addresses the context of self, partial and dynamic reconfiguration, in both its mono-dimensional and bi-dimensional cases. Self Reconfigurable Systems are completely independent in their management, thus they have the need to internally host reconfiguration management functionalities, such as core allocation, and to store or be able to autonomously obtain configuration bitstreams when needed. In Partial Reconfiguration, the functionality of single portions of the reconfigurable device can be modified, while the remaining parts of the device remain unchanged. Furthermore, with Dynamic Reconfiguration, the device portions that are not directly involved in the reconfiguration can continue their computation without interruption due to the reconfiguration process. The 1D reconfiguration paradigm only allows dynamical reconfiguration of columns spanning the whole device vertically, while the 2D paradigm gives the possibility of configuring fractions of the FPGA of arbitrary rectangular shape; the 2D approach is more powerful because of the added flexibility but is also more complex to manage. In such a context we will refer to a functionality that will be implemented on the reconfigurable device as core. Furthermore, as we allow the possibility for a core to have different implementations (i.e. with different shapes on the device), we will call Reconfigurable Functional Unit (RFU) each one of these implementations; with this definition, RFUs have a one-to-one correspondence with partial configuration bitstreams, except for what concerns the actual position on the device which may vary. Aim of this work is to support the creation and the management process of self partially and dynamically reconfigurable systems. In particular, but not exclusively, this goal includes automatic area constraint definition for cores, generation of efficient runtime core placement manager and the creation of solutions for internal and fast cores relocation. When exploiting partial reconfiguration, it is important to define reasonable area constraints for cores that are going to be configured inside the system. Area constraints definition, described in details in Section 3.1.1, impacts both on the performance of the cores and on their area usage, and both those aspects are critical for the global effectiveness of the reconfigurable system. This becomes even more critical when exploiting 2D reconfiguration as, instead of managing fixed height RFUs, the designer also has the freedom to examine various – rectangular – shapes. Since the considered scenario is that of self reconfiguration, the final system also needs to be able to autonomously perform choices during computation; this requires, in particular, an internal solution for core allocation management, which includes maintaining information on the reconfigurable fabric state and being able to choose where to place new cores upon their arrival. We refer to
these two aspects with the terms Empty Space Management and Online Placement Policy and to the global problem with the term Online Core Allocation Management. The main points of innovation for the proposed workflow are:

- Run time reconfiguration management: the flow has been defined to support the designer in the identification and in the creation of the best solution that can be used to manage at run time the partial reconfiguration process;
- Flexibility and Generality: the flow is designed to support different reconfiguration models, allowing the user to create a wide range of different reconfigurable architectures.
- Integration: the flow includes support for area constraints definition, core allocation management and relocation, all in an integrated environment.

2. RELATED WORK

In literature, several works concerning the defined problem can be found: all of those solutions, however, address only parts of the problem, the most common being empty space management, placement policies, or both, while several works can be also found for what concerns bitstreams relocation. For what concerns area constraints definition in a partially reconfigurable environment, currently there are no automated solutions; the problem is left to the designer skill in finding reasonable solutions that achieve a good compromise between area and performance. Online core placement is critical in dynamically reconfigurable systems, especially when exploiting 2D reconfiguration; a core can be placed in many different ways and bad placement can cause high fragmentation, which in turn can lead to loss of performance due to delayed cores. In addition to that, in self reconfigurable systems, the allocation manager runs inside the system, and thus directly impacts on its performance; for this reason, it is important to keep its overhead as low as possible, while still achieving good placement quality. A core allocation manager is composed of an empty space manager and a fitter. The former keeps track of the state of the configurable area, providing navigation through placement possibilities to the fitter which, in turn, has the role of effectively choosing which of the available locations is the best according to a placement policy. The most common placement policies that can be found in literature are: First fit (FF), Bottom Left fit (BL), Best/Worst fit (BF/WF), Best/Worst fit with exact fit (BFEF/WFEF), Fragmentation Aware [1] (FA) and Routing Aware [2] (RA).

The solutions proposed for empty space management can be roughly classified in two groups: those that keep complete information on the device state and those that heuristically prune it. The main strength of the first group is the guarantee that, if a suitable placement location exists, then it will be found. On the other hand, solutions that prune the information can use a considerably less amount of memory and provide a solution in a much faster time. Among the complete solutions, the KAMER[3] method represents the reconfigurable device state as a set of rectangles, one for each possible placement location and is based on general placement policies. It has an high complexity, quadratic in the number of placed cores, but it is valuable for its high placement quality. KNER[3] was created to reduce the complexity of KAMER while still achieving a reasonably good placement quality. The difference lies in which rectangles are considered: KAMER keeps all rectangles, while KNER only considers non-overlapping ones. With this simplification their number can be, at worst, twice the number of placed modules, giving a linear complexity. CUR[2] is based on a completely different approach: this method uses computational geometry to build a complete representation of the device state that can be explored in $n \log(n)$ time. The representation traces the global contour of all the clusters of modules that are currently placed on the FPGA. This solution is combined with a placement policy focused on minimizing inter-core distances and, thus, routing costs. Finally, the 2D-Hashing solution [4], is based on a hash table structure, that is accessed to find the placement for a Core. This approach exploits the fact that, considering a fitting criterion that only depends on core size or shape, the viable solutions can be precomputed during previous reconfigurations and, when a core arrives, the algorithm can simply check the table for the best location that is large enough, if any, in constant time. The amount of memory that is used is still linear in the number of placed cores. The main drawback of this solution is the impossibility to exploit focused placement policies.

3. THE PROPOSED APPROACH

The self reconfigurable architecture we are targeting is defined by a static area and a reconfigurable one; the reconfigurable area can be seen as composed of several reconfigurable regions and the reconfiguration of those regions, with the correct cores, is implemented using a general purpose processor embedded in the static region. The proposed approach includes both offline support, in the form of an automated Design Flow, described in Section 3.1 and runtime support, in the form of a solution for online Core Allocation Management and Relocation, described in Section 3.2.

3.1. Design Flow

The proposed design flow is structured in four main phases: input management, solution identification, system generation and bitstreams generation. Figure 1 shows a block diagram of the flow. The input management phase allows the user to provide all the necessary data, which can include
the specification of the target application, the information on the device that is going to be used to implement the system, the chosen reconfiguration model (1D or 2D) and the desired communication infrastructure. The target application is specified as a set of cores, each one representing a functionality that will be implemented with reconfigurable logic; the level of detail at which the application is specified can vary greatly: it can range from simply selecting a set of cores from a predefined library to fully defining a whole application, composed of customized cores. The solution identification phase is the part of the flow that performs the critical choices for the creation of the final architecture. First of all, the system tries to obtain a reasonable solution for area constraints definition and reconfigurable device partitioning, based on the distribution of cores given by the user, on the structure of communication channels, on the chosen model and on the device structure and size. In addition to that, the possibility of allowing more than one implementation per core is also considered; in particular, large cores will have, in general, more different implementations than small ones, to improve their chances at being successfully placed. Different placement policies for this first definition are then experimented, to choose the most suitable one for this specific scenario. Once both steps are complete, the most suitable relocation solution is added and the effectiveness of the resulting system is evaluated. This process is repeated many times, each one using the feedback from the previous ones. The most promising solution is then proposed to the user that can accept it, examine other found solutions or completely restart the process by giving some additional constraints or hints to the system. Once the solution has been identified, the system generation phase generates the components of the new reconfigurable system: the area constraints definition together with the partitioning of the device, an online Core Allocation Manager (provided in the form of processor code that will run inside the system) based on the chosen policy and a bitstream relocation solution. Finally, the bitstreams generation phase provides the generation of the actual implementation of the architecture. One of the files is the bitstream which defines the static part of the reconfigurable architecture, which includes communication infrastructure, processor, internal memory and relocation filter. This phase also generates all the partial bitstreams that define the various cores of the application.

3.1.1. Area Constraints Definition

An Area Constraint defines the shape of a core and can be specified by fixing its height, its width or both of them. The designer may constrain the shape of any number of cores of the application; the system automatically defines a set of shapes for all the other cores. The number of different shapes that are defined by the system for a core can be chosen by the user. Another parameter that can be set is the tightness of the system generated constraints set, which is a trade-off between area and performance. It is important to note that, to verify whether the implementation of a core with a given constraint is feasible, synthesis is needed. As synthesis is an expensive process, the default value for tightness will leave some margin to ensure feasibility, this at the price of some wasted area. The default value is empirically estimated to guarantee a near-certain feasibility but, if the user is willing to improve the area usage, it can be reduced, increasing the risk of unfeasibility. During the Solution Identification phase, Area Constraints are iteratively altered. The constraints given by the user are not changed, since they are considered to be strict requirements; however, if they are unfeasible, the flow will alert the user and ask for them to be changed (also providing a suggested value). In the first iteration, one shape per each Core is needed; in the case of user-defined constraints, that shape is used. Otherwise, a first heuristic implementation is defined with a criterion based on achieving a square-like shape that keeps relatively low area usage but, at the same time, near-certain feasibility. The formulae that are used to achieve this are:

$$H = \left\lfloor S \times (1 + m) \right\rfloor; \quad W = \left\lceil \frac{S + m}{H} \right\rceil$$

Here $H$ and $W$ are height and width of the constraint in slices, $S$ is the number of slices of the Core and $m$ is the margin that is left to keep a low risk of unfeasibility (an example value for this can be 0.15, which means 15% leftover). This first set of square-like constraints is then used to define the slot size and number for the reconfigurable system, unless already specified by the user; the formulae used to compute those values are the following:

$$\text{Rows} = \left\lceil \frac{Vg \times VSlices}{avgH} \right\rceil; \quad \text{Cols} = \left\lceil \frac{Hg \times HSlices}{avgW} \right\rceil$$

$$\text{SlotH} = \left\lceil \frac{Vslices}{\text{Rows}} \right\rceil; \quad \text{SlotW} = \left\lceil \frac{HSlices}{\text{Cols}} \right\rceil$$

where Rows and Cols are the number of vertical and horizontal slots that will be created, VSlices and HSlices are the height and width of the FPGA in slices, avgH and avgW are the average height and width of the Cores in slices (which

---

**Fig. 1. Design flow overview**
are available at this point, either because given by the user or computed using the previous formulae); Vg and Hg are user-defined integer parameters that express Vertical and Horizontal granularity (the higher the values the smaller slots will tend to be, for example if Vg is 1 and Hg is 3 the average core will span 1 row and 3 columns) and, finally, SlotH and SlotW are the height and width of a single slot in slices. In the subsequent iterations of the Solution Identification phase, the constraints are altered. This can happen in two ways: if the user allowed more than one implementation for that core, new shape options are added to that core, if only one implementation is allowed the existing one is altered (if it was not defined by the user). The alteration of the shapes is performed by either increasing the height of the core by 1 slot at a time and adapting the width or the opposite; if more implementations can be kept, both kinds of alteration can be performed at the same time.

3.2. Runtime Support

This section presents the proposed solutions for runtime support of the 2D self partially and dynamically reconfigurable architectures.

3.2.1. Core Allocation Management

The proposed approach to cope with the online core allocation management is an improved version of the work proposed in [5]. The base solution that was developed is built upon an efficient empty space manager that keeps a reduced amount of information on the device state to reduce memory usage and search time. This is because the solution is going to be executed on the reconfigurable device hardware, impacting on its performance, thus we chose to limit this negative impact by using a low-complexity solution. This base solution, however, is designed to be flexible and to support different scenarios and customized placement policies. In particular, the scenarios we consider include: dynamic schedule scenario, blind schedule scenario and variants. In a **dynamic schedule** scenario core allocation requests arrive at an unpredictable time and with an As Soon As Possible (ASAP) and As Late As Possible (ALAP) time to be satisfied; the core allocation manager must then try to satisfy all the requests while satisfying the timing constraints. In this scenario, core allocation failures are a possibility and will tend to be, for example if Vg is 1 and Hg is 3 the average core will span 1 row and 3 columns) and, finally, SlotH and SlotW are the height and width of a single slot in slices. In the subsequent iterations of the Solution Identification phase, the constraints are altered. This can happen in two ways: if the user allowed more than one implementation for that core, new shape options are added to that core, if only one implementation is allowed the existing one is altered (if it was not defined by the user). The alteration of the shapes is performed by either increasing the height of the core by 1 slot at a time and adapting the width or the opposite; if more implementations can be kept, both kinds of alteration can be performed at the same time.

3.2.2. Bitstream Relocation

Relocation is a powerful technique that can reduce the amount of memory required to store the partial reconfiguration bitstreams and that can be implemented on the nowadays non homogeneous FPGAs [6]. At design time it is important to assign to each module a placement on the physical device. Unfortunately, due to runtime conditions, it might happen that the area assigned to a core can be already occupied by a different component. In such a scenario, without the relocation, we cannot serve the request for the desired core until the already mapped one ends its execution. To overcome this problem the proposed design flow includes a bitstream relocation technique based on the **blank module approach**, [7]. The approach is based on a bitstream, the **blank module** cm, that can be used to erase any configuration ci on a known location. In order to provide the automatically generated reconfigurable system with the relocation capability, we use in this work two main hardware solutions: a hardware relocation filter designed for the 1D scenario on Virtex, Virtex-E, Virtex-II, and Virtex II Pro devices, which is presented in [8] and another hardware relocation filter that can be used for 2D relocation on Virtex-4 [9] and Virtex-5 [10] devices. The two solutions have a similar global structure, but they are different in the implementation of their logic, due to the different target device, and the different reconfiguration scenario for which they are designed.

4. EXPERIMENTAL RESULTS

This section presents a set of experimental results related to the runtime aspects of our proposed approach, such as the
runtime core placement strategy, the core allocation management and the relocation technique. The first part proposes a performance comparison of a version of our solution with a policy based on minimizing routing paths between cores, w.r.t literature solutions. Then the experimental results related to the cores allocation management are presented. Finally, we describe the relocation experimental results that have been obtained in different scenarios, such as 1D and 2D, and analyzed w.r.t. the area usage, the maximum frequency and the maximum throughput.

4.1. Placement Experimental Results

4.1.1. Comparison with similar literature solutions
Table 1 shows the compared results obtained by the proposed solution integrated with a routing cost minimization fitting policy against the two that are most similar to it: CUR [2], which shares the fitting strategy based on minimizing routing paths but is not heuristic, and KNER [3] which follows the same philosophy of the proposed solution in empty space management but does not apply any focused fitting strategy. The benchmark used to perform this first evaluation is a set of 100 randomly generated cores, with size varying from roughly 5% of the reconfigurable device area to approximately 25% of it; the required interconnections were also randomly generated along with the cores and the routing cost obtained by each solution was subsequently computed with manhattan distance metric. The Core Rejection Rate reflects the percentage of all the cores that were not successfully placed due to suitable area not found by the allocation manager. The benchmark was run several times with different random sets and the results presented in the table reflect the average of the obtained results.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Running Time (per Core)</th>
<th>Routing Cost (average)</th>
<th>CRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUR</td>
<td>Complete, Routing Focused</td>
<td>156 ms (1.56 ms)</td>
<td>0.276 (2.27)</td>
<td>5%</td>
</tr>
<tr>
<td>Proposed solution</td>
<td>Heuristic, Routing Focused</td>
<td>51 ms (0.506 ms)</td>
<td>0.3 (3.6)</td>
<td>4.7%</td>
</tr>
<tr>
<td>KNER</td>
<td>Heuristic, Unfocused (FF)</td>
<td>39 ms (0.386 ms)</td>
<td>2.25 (11.4)</td>
<td>4.2%</td>
</tr>
</tbody>
</table>

4.1.2. Completion times when varying device size
This section presents a comparison of the application completion time with different amount of resources and different core densities. The scenario that is considered here is a blind schedule one, with variable arrival times for cores. The measure that is computed is the number of time instants that pass from the arrival of the first core to the completion of the one that was placed last; each core has an execution time, labeled latency. This experiment also provides a useful comparison w.r.t an infinite resources scenario, represented as the ratio between the completion time on a limited device and the completion time with infinite area. The cores that are used for this experiment are real world ones, taken from OpenCores.org and whose area constraints have been defined automatically, using the method described in the previous sections. In particular, the cores that have been used as benchmark include common applications such as 3DES, AES, JPEG, Cordic and a floating point unit. The device sizes that were considered are, in slots, of 4 rows by respectively 8, 10 and 14 columns. The slot size, defined again using our automated approach based on the distribution of core sizes in the application is of 45 slices (height) by 20 slices (width). To give an example, with those definitions, a JPEG core can fit in a 3 columns by 2 rows rectangle, while an AES can fit in 1 row and 2 columns.

4.2. Cores Allocation Experimental Results
This section presents a set of experimental results to evaluate the benefit of allowing multiple shapes per core in a Core Allocation Management solution. The experiment compares the results obtained by KNER [3], a literature solution for fast core placement that only allows a single fixed shape for each core, with those obtained by our solution, here called MSLP - Multiple Shapes Linear Placer, with 3 and 5 shapes per core. The goal of the experiment is to demonstrate that the cost increase, in term of number of bitstreams to be stored in the system and running time of the algorithm, is adequately rewarded by a significant improvement in placement quality, measured as Core Rejection Rate. The scenario we are considering here is, similarly to the previous one, a blind schedule one, with no dependences between cores. However we introduce a difference: in this experiment a deadline is associated to each core, if the core is not successfully placed before its expiration it must be rejected; this permits to actually compute a CRR value for the application.

<table>
<thead>
<tr>
<th>Application Size</th>
<th>Average latency</th>
<th>KNER (1 Shape)</th>
<th>KNER (3 Shapes)</th>
<th>KNER (5 Shapes)</th>
<th>MSLP (1 Shape)</th>
<th>MSLP (3 Shapes)</th>
<th>MSLP (5 Shapes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 cores</td>
<td>0.561</td>
<td>2.24%</td>
<td>0.259</td>
<td>0.3</td>
<td>0.316</td>
<td>0.316</td>
<td>0.316</td>
</tr>
<tr>
<td></td>
<td>0.273</td>
<td>3.8%</td>
<td>0.289</td>
<td>1.6%</td>
<td>0.311</td>
<td>1.6%</td>
<td>1.6%</td>
</tr>
<tr>
<td></td>
<td>0.255</td>
<td>6.7%</td>
<td>0.311</td>
<td>3.4%</td>
<td>0.318</td>
<td>3.4%</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Table 2. Single Shape vs Multiple Shapes using First Fit
5. CONCLUDING REMARKS

Aim of this work was the creation of a complete workflow to help the designer in the creation and runtime management of self partially and dynamically reconfigurable systems. The proposed solution represents a good compromise between complexity and quality of the results. The core allocation manager is highly versatile, supporting various kinds of fitting strategies and, in the focused version, combines the runtime advantage of the heuristic approach with the interconnection cost minimization advantage offered by routing aware focus. The core allocation algorithms are simple enough, both computationally and memory wise, to be run on the internal processor of reconfigurable systems. Among other aspects, the provided support includes the definition of area constraints for cores, the creation of an efficient runtime solution for core allocation management and the generation of a solution to obtain internal and fast relocation of cores.

6. REFERENCES