Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 942930, 10 pages
doi:10.1155/2009/942930

Research Article

A Design Technique for Adapting Number and Boundaries of Reconfigurable Modules at Runtime

Thilo Pionteck, Roman Koch, Carsten Albrecht, and Erik Maehle

Institute of Computer Engineering, University of Lübeck, 23538 Lübeck, Germany

Correspondence should be addressed to Thilo Pionteck, pionteck@iti.uni-luebeck.de

Received 30 November 2008; Accepted 17 May 2009

Recommended by Michael Huebner

Runtime reconfigurable system-on-chip designs for FPGAs pose manifold demands on the underlying system architecture and design tool capabilities. The system architecture has to support varying communication needs of a changing number of processing units mapped onto diverse locations. Design tools should support an arbitrary placement of processing modules and the adjustment of boundaries of reconfigurable regions to the size of the actually instantiated processing modules. While few works address the design of flexible system architectures, the adjustment of boundaries of reconfigurable regions to the size of the actually instantiated processing modules is hardly ever considered due to design tool limitations. In this paper, a technique for circumventing this restriction is presented. It allows for a rededication of the reconfigurable area to a different number of individually sized reconfigurable regions. This technique is embedded in the design flow of a runtime reconfigurable system architecture for Xilinx Virtex-4 FPGAs. The system architecture will also be presented to provide a realistic application example.

1. Introduction

Runtime partially reconfigurable FPGA devices like those of the Xilinx Virtex-4 and Virtex-5 series allow system designers to reuse hardware resources over time. Individual modules can be replaced at runtime so that only currently needed processing units (PUs) have to be instantiated in the FPGA. System designs making use of this feature typically comprise only one or few reconfigurable regions (PR regions) with fixed sizes, and locations, allowing for only a fixed number of reconfigurable PUs. For such systems, no special system architectures or design approaches are needed. The system structure is static while only PUs with fixed links are exchanged at runtime. Design tools such as Xilinx early access partial reconfiguration (EAPR) tools [1] and recent versions of the PlanAhead [2] floorplanning tool provide reasonable support for such designs.

Yet, as soon as the number, sizes, and locations of PR regions in the reconfigurable system partition are not static anymore, system layout becomes complicated. With varying communication needs of a changing number of PUs of different sizes mapped onto diverse locations, the system architecture has to provide mechanisms to back this flexibility. This feature is not provided by commonly used system architectures based on buses or point-to-point connections for the communication network. These kinds of communication infrastructures require that all possible combinations of PUs have to be determined at design time so that the communication infrastructure can be dimensioned to support all possible scenarios, resulting in a huge hardware overhead. In addition, current design tools put tighter restrictions on system layout than imposed by the actual hardware. The number, sizes, and locations of reconfigurable regions have to be fixed at an early stage in the design process. There are no provisions for adapting this spacial partitioning later. As a consequence, each PR region has to be dimensioned according to the footprint of the largest PU. Such an approach results in a potential waste of logic resources and is not satisfactory as it is technically well feasible, for example, to move the border between two neighbouring regions in order to provide more space for the one module if the other is currently not needed or requires less resources.

To solve these issues, a system architecture for runtime reconfigurable designs based on Xilinx Virtex-4 FPGAs is

proposed. The reconfigurable system partition is divided into tiles which can be replaced individually at runtime. PUs mapped onto these tiles are connected by a topology adaptive network-on-chip (NoC). The boundaries of the tiles within the reconfigurable partition can be adjusted at runtime, allowing an area efficient mapping of PUs of unequal complexity. While the basic principles of this system architecture were already presented in [3], the design methodology for exploiting the features of the system architecture has not been available till now. This contribution closes this gap by presenting a self-contained technique that allows to change the boundaries and number of PR regions using the available tools, yet circumventing their restrictions. The technique will be demonstrated by means of a comprehensible example, and it will also be shown how this technique facilitates the design of adaptive runtime reconfigurable systems.

The rest of the paper is structured as follows. Section 2 discusses related work in the area of runtime reconfigurable system architectures for designs with a variable number of exchangeable PUs. A brief overview of the proposed system architecture is given in Section 3 and provides the motivation and context for the design technique presented in Section 4. Section 5 demonstrates the advantages when applying the design technique to runtime reconfigurable systems, and, finally, Section 6 summarises the contribution of this paper.

2. Related Work

In [4, 5] an architectural template for runtime reconfigurable systems is presented. The idea bases on algorithmic skeletons which provide a separation of the structure of the computation from the computation itself. Algorithmic skeletons were first introduced in the 1980s by [6] and were extended to dynamic reconfigurable computing in [4]. Here, they act as a kind of abstraction layer on top of an FPGA making the concept independent from a specific FPGA. The reconfigurable fabric is divided into tiles of equal or unequal sizes which are used by a skeleton dispatcher for the hardware realisation of algorithmic skeletons. Algorithmic skeletons are written in VHDL and are enfolded with a wrapper that handles all communications. Mapping the algorithmic skeletons to tiles is done manually by a pattern designer. The pattern designer also has to guarantee that enough communication resources between tiles are available so that point-to-point communication between skeletons is possible as well as communication with a central storage unit. The partitioning of the FPGA in tiles is done at design time and cannot be adapted at runtime.

The idea of adapting the size of PR regions to the size of the actually implemented PU was also picked up in [7]. At design time, a Virtex-II FPGA is partitioned into fixed sized configuration slots, each adjacent to a routing channel. The configuration slots are used at runtime to implement the PUs. In case of a PU not occupying the complete height of a configuration slot, additional PUs with the same width may be mapped onto the same slot. This is done by online adapting the addresses of the configuration data of the PUs at the granularity of CLBs. Connections between PUs and the static part of the system are provided by routing channels. Each routing channel is connected to the static system and consists of vertical links. In case a PU has to be connected to the communication network, the vertical communication links at the position of the PU interface are exchanged by special communication primitives providing horizontal and vertical links. In order to ease online routing, each PU has an LUT-based communication interface at a predefined position, for example, in the lower right corner of the PU. The design flow required to build up such a system is described in [8] and bases on Xilinx JBits class library [9].

A design methodology for generating on-chip communication infrastructures for partially reconfigurable FPGAs with a variable number of PUs is also presented in [10]. The PR region is partitioned into several homogeneous tiles which all provide identical links to the communication network. This allows the mapping of PUs onto any set of tiles in the PR region. The mapping itself is determined either at design time or at runtime by system specific placement algorithms. Address mapping of PUs to tiles of the PR regions is handled transparently so that a position-independent communication can be guaranteed. The communication infrastructure is divided into five abstraction layers, simplifying system adaption to the resources and reconfiguration capabilities of a specific FPGA.

In [11, 12] COMMA, a methodology for automatically generating a communication infrastructure for systems targeting Virtex-4 FPGAs, is introduced. The system layout consists of fixed sized PR regions surrounded on the left and right side by slots reserved for the communication infrastructure. The communication infrastructure is determined at compile time by analysing the communication requirements between PUs based on a given application. An ILP-based approach is used to assign PUs to PR regions so that the number of wires in the routing channels is reduced as well as the communication delay. Communication between adjacent PUs in the same column is done without accessing the communication network. PUs may span adjacent PR regions, and it is also possible to implement several smaller PUs on one PR region. Reconfigurable data ports are used to connect the I/O pins of the PUs to the communication infrastructure at reconfiguration time.

An early work providing a system infrastructure and design methodology to set up runtime reconfigurable systems with an arbitrary number of PUs to be exchanged at runtime is presented in [13]. Targeting the Xilinx Virtex-E FPGA series, the system is divided into a static infrastructure and a number of so-called Dynamic Hardware Plugins (DHPs). The static infrastructure is routed in such a way that no nets cross any DHP regions. This is achieved by a modified version of the router. DHPs are implemented in separate designs by using a gasket interface which on the one side provides fixed interconnect points to communicate with the static infrastructure and on the other side prevents logic from being placed and nets from being routed outside a dedicated area. A special tool called PARBIT is used to extract the configuration code for the DHP from a bitfile and to retarget it into a similar sized region of the FPGA in the final system.

The main difference between the related work and the system architecture and design technique presented in this paper is that here tiles and, hence, PR regions can either be used to implement PUs or components of the communication network. There is no need to provide dedicated routing channels next to PR regions, and PR regions can directly adjoin each other. System connectivity is provided at a level of tiles by mapping components of the communication network onto tiles. In addition, the initial partitioning of the reconfigurable system partition into PR regions can be adapted at runtime. Thus, the presented architecture offers two degrees of freedom for the system layout: mapping of PUs onto different PR regions and resizing of PR regions at runtime.

3. System Architecture

The basic structure of the runtime reconfigurable system architecture is depicted in Figure 1. This architecture takes into account the reconfiguration capabilities and physical restrictions of the Xilinx Virtex-4 FPGA series. Yet, its principal ideas are device independent. A detailed description of the architecture is given in [3]. Here, a brief summary is given in order to provide the background and motivation for the proposed design technique.

The FPGA logic area is divided into a static and a reconfigurable partition which in turn is subdivided into a grid of equally sized tiles. The reconfiguration control logic including the internal configuration port (ICAP), I/O controller, system controller, and all static PUs resides in the static partition. PUs to be exchanged at runtime as well as the communication network to interconnect them are realised onto the grid of tiles in the reconfigurable partition. For the communication network a topology adaptive packet-based NoC called Configurable Network on Chip (CoNoChi) [3, 14] is used. CoNoChi comprises 16-bit wide virtual cut-through switches with four full-duplex links. Routing is done by means of locally stored routing tables supplied by a global control unit. The NoC protocol supports different priority classes for sending routing tables, control messages, and data.

Four different types of basic building blocks can be mapped onto the grid of tiles in the reconfigurable partition: horizontal and vertical communication links, network switches, and application specific blocks, for example, PUs. Each block provides, at the same position, communication links to adjacent blocks so that each block can be connected with the other. The communication links consist of so-called bus macros which are communication lines crossing the border between two blocks with endpoints routed to LUTs on either side of the border. At runtime, blocks mapped onto tiles can arbitrarily be exchanged, allowing to set up a random NoC structure and arrangement of PUs. Each tile consists of 12×16 Configurable Logic Blocks (CLBs) and is aligned to the reconfiguration capabilities of the Xilinx Virtex-4 FPGA series. While this size fits well with the resource requirements of a CoNoChi switch of about 120 CLBs, PUs may require to combine the resources of several tiles. From the hardware point of view, this is feasible. However, the design flow described in [1] to implement partially reconfigurable designs thwarts this approach. Based on [1], the boundaries of any individually reconfigurable region have to be fixed exactly once at design time. Thus, an adaption of the boundaries of tiles is not possible afterwards, in particular, not at runtime. The design technique described in the next section circumvents this constraint.

4. Adapting the Number of PR Regions

The design technique bases on Xilinx EAPR tools and also uses the PlanAhead floorplanning tool. Compared to the former design techniques for building runtime reconfigurable systems [15], the EAPR tools significantly ease the design process and impose fewer restrictions on the system layout. In the context of this paper, the main advantages of the EAPR tools are that PR modules no longer have to cover the whole column of an FPGA and that signals of the static components may cross PR regions without the use of bus macros. The first feature allows with some restrictions to define PR regions of almost arbitrary shape and, thus, an adaptation of the PR region to the size of a PU. As I/O signals of the PR region have to be routed through bus macros spanning the boundary between PR region and adjacent region, one PR region cannot be resized without adapting the adjacent region. In order to omit online placement and routing, all possible scenarios of PR locations and sizes for a given system have to be identified at design time. Relating to the system architecture described before, this results in the necessity to generate separate configuration files for all different tile locations and tile types. In the EAPR flow, the boundaries constraining the PR regions must be defined once at the beginning. Hence, the entire flow has to be run again when number, size, or locations of PR regions are changed. Moreover, different numbers of PR regions require different numbers of bus macros and, thus, result in different top-level netlists. Therefore, the proposed merging and separating of PR regions at runtime makes it necessary to combine the results of different runs of the EAPR tool flow. This leads to an issue with static nets through PR regions which might be routed in different ways in multiple, independent runs of the flow. Functional failures or even short-circuits may then occur when merging or separating PR regions by reconfiguration. Thus, a technique has to be set up which guarantees that these static routes are consistent across various runs of the tools.

4.1. Technical Background. The actual implementation process for partially reconfigurable systems is divided into several phases. Initially, rectangular regions in the FPGA grid are defined for the PR modules in order to constrain routing and logic into the corresponding region. After this, bus macros are locked to positions on region boundaries to allow boundary-crossing communication. A *static design implementation* is then built from the top level design while leaving out the PR modules. Subsequently, the PR modules are implemented individually, again including the top level design. Finally, a desired set of PR module implementations

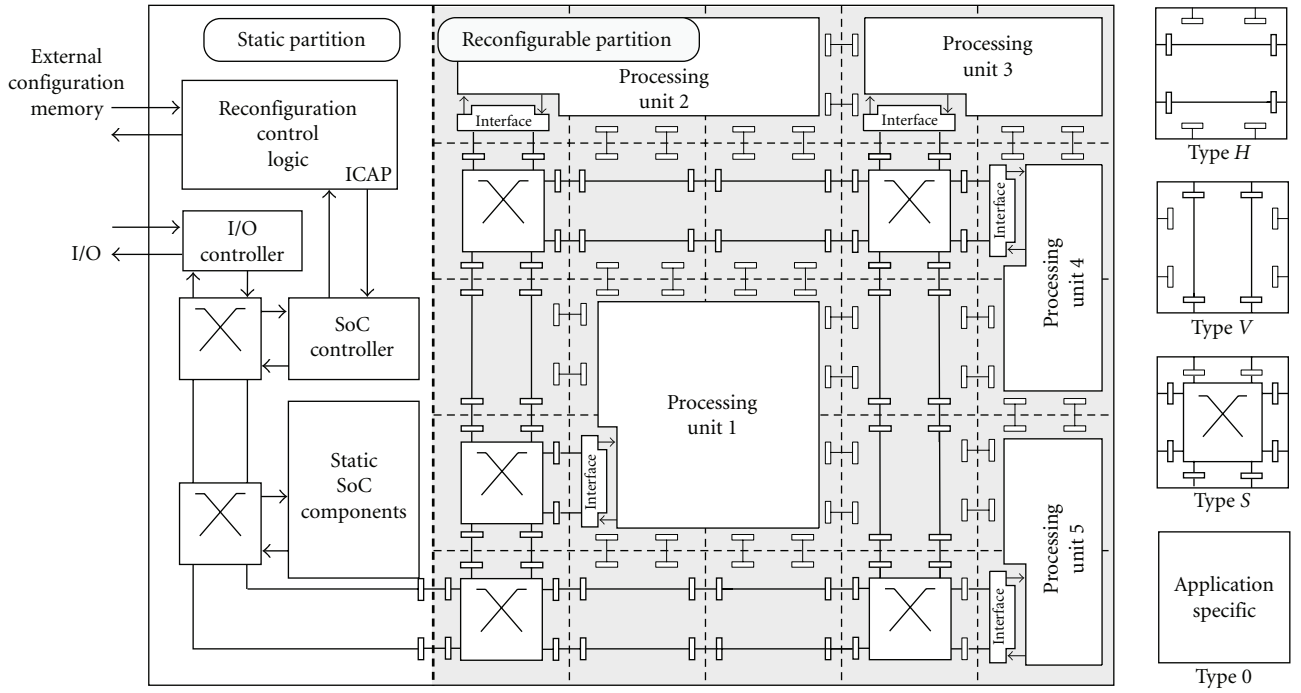


Figure 1: System architecture [3].

is merged into the static design implementation. Configuration bitstream files are generated for the complete design as well as for partially reconfiguring individual PR modules.

Ideally, any static logic and routing is kept out of the PR regions. In practice, however, there can be externally given constraints that make it necessary to place, for example, an input buffer for a signal entering the FPGA inside such a region. The EAPR tool flow manages these situations by keeping track of which FPGA resources are occupied during the static design implementation phase. These resources are then prevented from being used again by PR modules so that there are no conflicts when it comes to merging static and reconfigurable implementations.

These precautions taken by the tools, however, are only effective within a single scenario. Let S_k be a scenario characterised by a certain combination of a top-level design, static modules, and a floorplan of PR regions. Changing the number or sizes of PR regions means switching from S_k to S_l ($k \neq l$) and, thus, requires the top-level design and the static modules to be reimplemented. The resulting static design implementation of S_l may use different resources within PR regions than does its counterpart from S_k. Hence, the PR modules built within S_l are likely to be incompatible with S_k. Checks within the tools prevent straight-forward approaches like trying to reuse the resource exclusion information from S_k for S_l from being successful, and so a different solution must be found.

4.2. Proposed Design Technique. The proposed technique for exactly matching static routes in PR regions of different projects for the same system is illustrated in Figure 2. Assuming two or more modular PR designs that share the same static elements and define the same interface between the static and the reconfigurable parts, the basic idea is to first implement the common static elements in the context of one design and then to transfer the result into a *hard macro* for replacing the static elements defined in the other design. A hard macro is a completely prerouted and preplaced block of logic circuits and interconnects. A hard macro can be instantiated at HDL level multiple times and can be locked to different places inside the FPGA. Hard macros allow parts of a design to be reproduced in exactly the same way throughout different versions of the design.

As the static elements of the different scenarios were functionally identical before and as all static elements are resembled by the hard macro, this approach keeps the affected design consistent. The technique comprises five steps and operates on netlists of the modular PR designs created with standard tools. Each step will be explained in the following by means of an illustrative example implemented on a Xilinx Virtex-4 FX60 FPGA. The example is shown in Figure 3 and consists of two different scenarios. The static part of the design comprises external I/O connections, circuitry for receiving and sending data, and a control unit. In addition, there are four different types of PUs: A *NULL* module just forwards the incoming data, an *XOR* module calculates an exclusive-or on the data, a *DES* encryption module gathers 64 bit of data and then performs a DES encryption, and an *AES* module which collects and encrypts 128 bit. As the AES module requires more resources than the other modules, the design comes in two variants. The dark shaded modules in each PR region of Figure 3 make up a scenario S_1 while the lighter shaded modules correspond

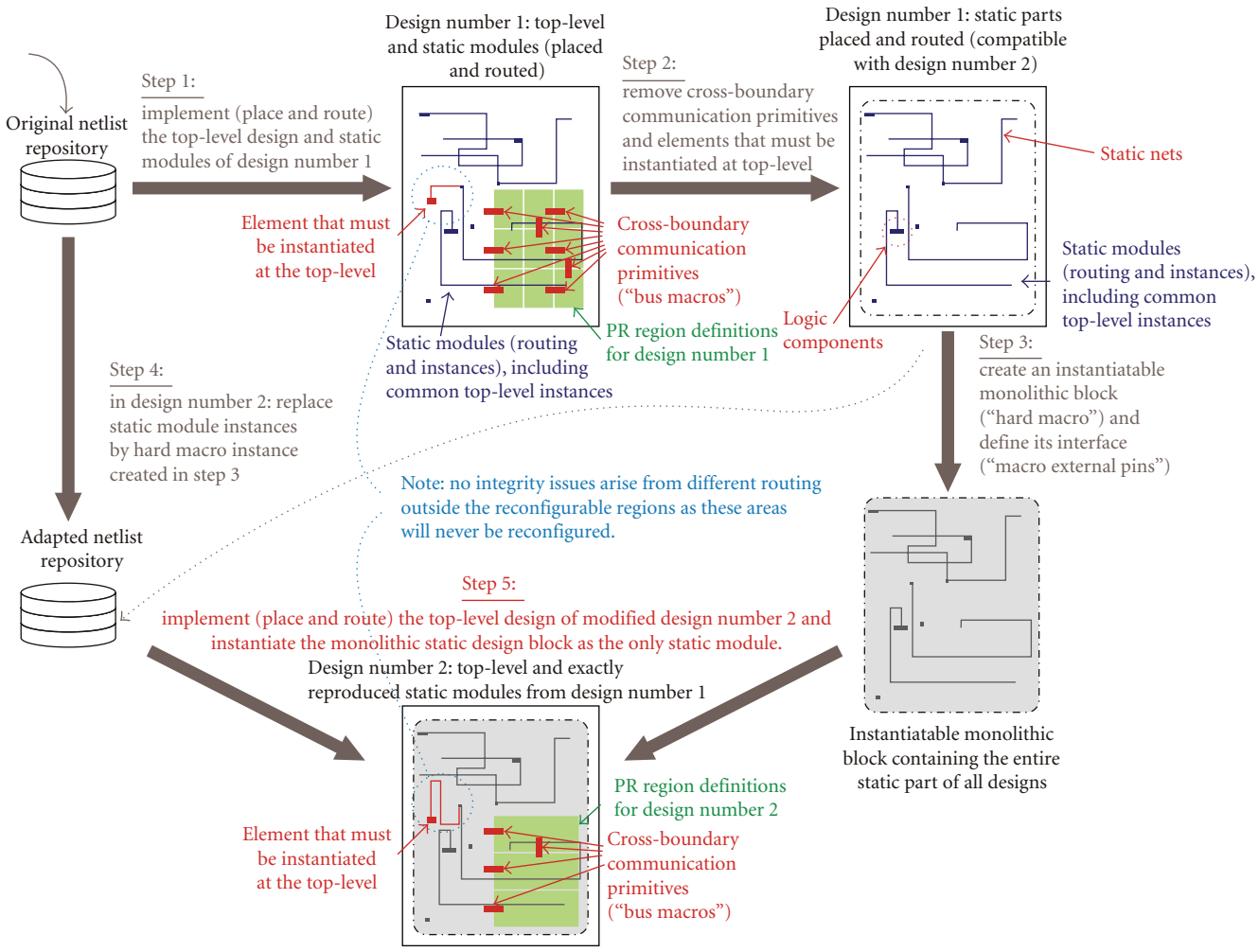


Figure 2: Exact cross-project reproduction of static design artifacts within PR regions.

to scenario S_2. Thus, the first scenario (S_1) defines four equally sized PR regions, and the second (S_2) defines three PR regions. In S_2 the two upper regions are combined to one in order to provide space for the larger AES module implementation. The corresponding floorplans created with PlanAhead are shown in Figure 4.

Step 1 (implementing the static parts of scenario S_1). Seen from the outside, the top-level design provides all the external ports necessary for communicating with the design, including a clock input. Internally, the top-level design instantiates a single static module which contains all the static components of the design. At the same hierarchy level as this static module, the four PR modules of scenario S_1, several bus macros, I/O buffers for external connections, and clocking primitives are instantiated. As, later on, the static part of the design will be extracted as a hard macro that has to be connected to the bus macros for the communication with the PR regions, precautions are taken not to "loose" the pins of the static module. All signals between the static module and bus macros are routed through explicitly instantiated FPGA primitives—1:1 look-up tables (LUT1) programmed with the identity function. These LUT1 instances are constrained to exact locations close to the bus macros. So, signals from and to static modules become available at exactly known locations in the FPGA. Figure 5 gives a close look on the actual routing between the static design and bus macros residing on the boundary to a PR region. One end of the corresponding nets is connected to the bus macros while the other end is connected to LUT1s which are part of static design. The implemented design is shown in Figure 6. It does not include any PR module implementation as these modules were considered as yet unimplemented black boxes.

Step 2 (removal of primitives instantiated at top-level). In order to turn the static design into a re-instantiatable hard macro, the static design has to be adapted in FPGA Editor. This tool is first used to save routing information of nets which must be included in the hard macro, for example, clock nets. As hard macros cannot contain these routing

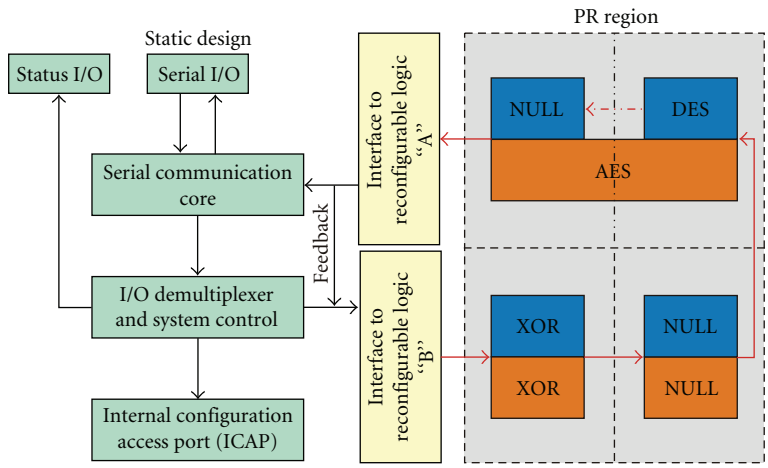
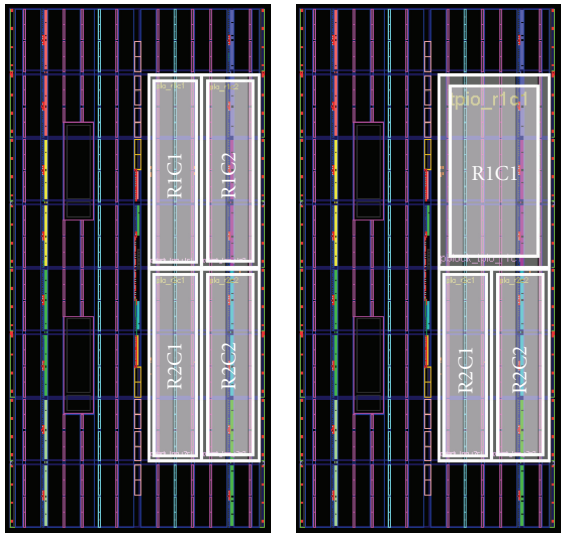


Figure 3: Example design.



(a) scenario S_1: four equally sized PR regions

(b) scenario S_2: upper two regions combined to one

Figure 4: Two scenarios (PlanAhead floor-plans).

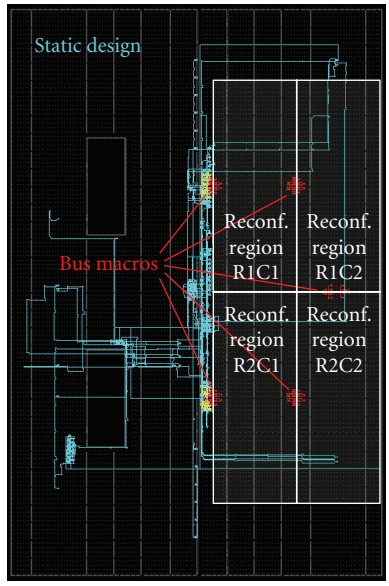


Figure 6: Static design intersecting PR regions.

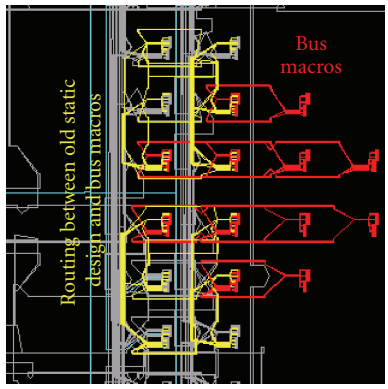


Figure 5: Routing between static design and bus macros.

information, data have to be saved in a separate file to reproduce routing of affected nets later during the design process. In a next step, top-level elements such as bus macros and clocking primitives are removed from the design. These elements also must be immediate children of the top-level of scenario S_2 and, thus, must not be added again to scenario S_2 when including the hard macro. An example of such an element is given in Figure 2. The element encircled in the floorplan after Step 1 is removed in Step 2. As the net connected to this element crosses the PR region and, thus, has to be part of the hard macro, the net is segmented by routing it through an explicitly instantiated FPGA primitive which is part of the hard macro.

Step 3 (creating a static hard macro). The remaining design consists of modules and nets of the static part of scenario S_1. These elements are also part of scenario S_2. In order

to replace these elements later in scenario S_2, the design is converted into a hard macro, and "external pins" are assigned for the clock input and for the known locations of signals to and from the PR regions. In the following, this hard macro will be referred to as the *static hard macro*.

Step 4 (implementing the static parts of scenario S_2). In contrast to the top-level design for S_1 the top-level design for S_2 is heavily thinned out; for example, the only external port of the new design is a clock input. There are no other external ports, because all I/O buffers of the system are incorporated into the static hard macro. Besides the static hard macro, the new top-level design instantiates bus macros as needed for scenario S_2, three—instead of four—PR modules, and mandatory top-level elements.

Step 5 (merging scenario S_2 and static hard macro). In the final implementation phase, the saved routing information from scenario S_1 is utilised to reproduce routing, for example, of clock nets. The resulting design is shown in Figure 7. Note that the routing of nets from elements which have to be instatiated at top-level may differ between scenario S_1 and scenario S_2. As these nets have been segmented before by routing through explicitly instantiated FPGA primitives, scenario S_1 and scenario S_2 provide identical links to these nets. Hence, it does not matter whether the routing from these fixed links to the elements differs in both designs. Only one version will be implemented in the final design, and the static area will never be reconfigured.

Finally, PR modules are merged with their respective static designs. Partial bitstreams are created for individually reconfiguring the PR regions. Most importantly, it is considered safe to dynamically load PR modules of either scenario into an FPGA initialised with the other. Note, however, that it might be necessary to blank a region beforehand in order to avoid short circuits. Figure 8 shows both scenarios fully implemented. For switching between the scenarios only PR regions R1C1 ($S_{1,2}$) and R1C2 (S_1) need to be reconfigured.

4.3. Pros and Cons. Although the proposed technique intercepts the standard design flow multiple times, it completely bases on Xilinx tools and does not require any additional tools manipulating the configuration bitfiles. One drawback of the proposed technique is that the placed and routed design has to be edited manually in FPGA Editor. Yet, once the static hard macro is created it can be reused easily in numerous scenarios that share the same static design. In principle, there also exist two alternative approaches for resizing the boundaries of PR regions, though each approach has significant drawbacks compared to the proposed technique.

4.3.1. Avoidance of Static Nets in PR Regions. Static routing that intersects PR regions is most often caused by I/O buffers residing inside these regions. In order to *avoid* such routing, the I/O buffers could alternatively be instantiated inside the PR modules instead of the top-level or static module [16, 17]. The signals would then have to be passed through additional

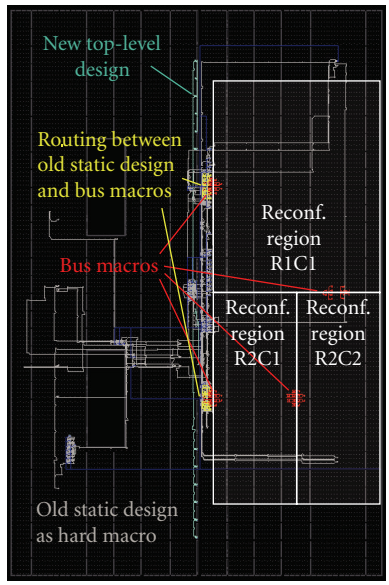


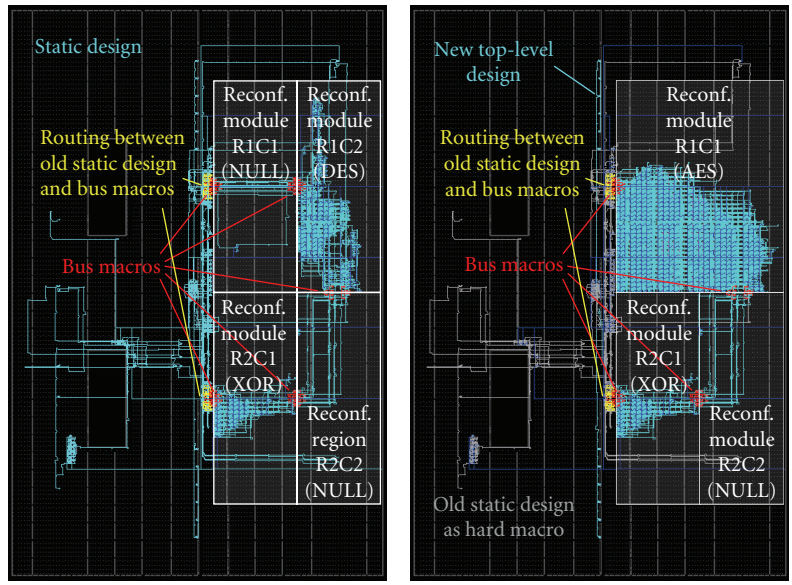
Figure 7: Static design of S_2 with static design from S_1 instantiated as a hard macro.

bus macros, potentially through multiple PR regions. Under the tool flow aspect, this approach is very clean. The insertion of additional bus macros, however, is likely to turn out even more cumbersome than creating a static hard macro as proposed in this paper. Furthermore, owing to the fact that signals are routed in PR modules, a single signal may be routed in various ways through the same PR region and, thus, show varying propagation delays. Even worse, such a signal can be interrupted during reconfiguration. Most important is, however, that it can hardly be guaranteed that really no static routing at all intersects PR regions. Consequently, this approach is not safe to rely on.

4.3.2. Directed Placement and Routing of the Static Design. Instead of using a hard macro, the static design could also be exactly reproduced for different scenarios by utilising "directed routing". Following this approach, the static design would also be opened in the FPGA Editor. Complete routing and placement information would then have to be exported in a user constraint file (UCF). This UCF file would be included into subsequent static design implementation runs. The handicaps of this approach are that potentially very large text files have to be handled and that it is necessary to reimplement the whole static design for every scenario.

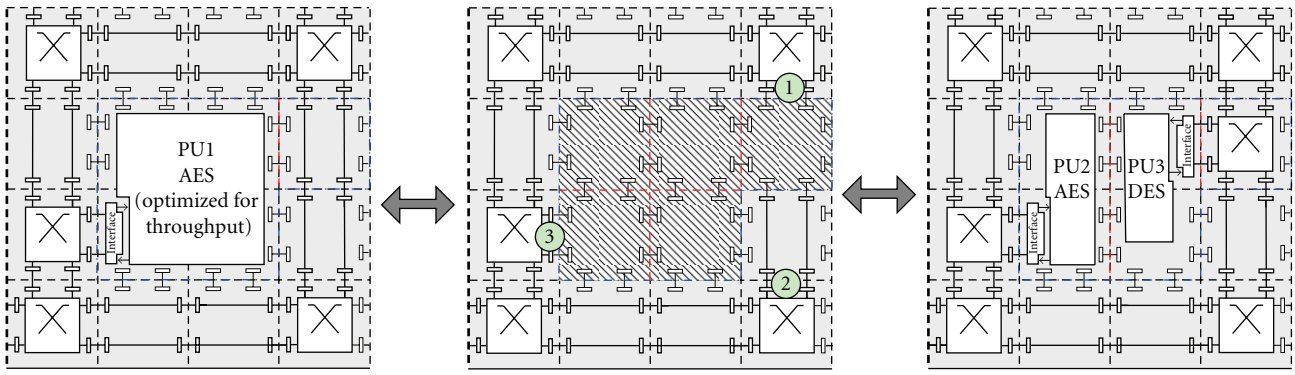
5. Application Example

The design technique was developed in the context of the system architecture presented in Section 3. This architecture will now be used to demonstrate the usefulness of this technique by means of an application oriented example, a coprocessor platform for network processors called Dynamically adaptable Coprocessor based on Reconfiguration (DynaCORE) [18].



(a) scenario S_1 (b) scenario S_2

Figure 8: Complete designs (PR modules merged in static design).



(a) Scenario S_1: one AES module (b) Intermediate step: blank tiles (c) Scenario S_2: two small encryption cores

Figure 9: Application example for changing boundaries of PU regions in the context of DynaCORE.

DynaCORE is an adaptable hardware accelerator for increasing the performance of network processors when deep packet processing is required. Typical deep packet processing tasks comprise compression, network-intrusion detection, or encryption/decryption, for example, when virtual private networks are to be established. As network processors are primarily optimised for header processing, the increasing demand in deep packet processing tasks asks for dedicated accelerators. These accelerators have to be flexible as network traffic composition and processing requirements vary throughout the day. DynaCORE provides this flexibility by autonomously determining an optimal set and arrangement of PUs according to the actual traffic profile.

A typical scenario for a dynamic exchange of PUs of different sizes within DynaCORE is depicted in Figure 9. Figure 9(a) shows the mapping of a fast and hardware intensive AES core onto four tiles. These four tiles are combined to one PR region and only provide bus macros at the boundary of the region. The complete logic area inside the PR region is used to implement the AES core. Compared to a partitioning of the AES core into four separate components to be mapped onto four individual PR regions, the problem of partitioning the core into components with only few links between them is avoided, no bus macros are required to link the components among each other, and the cross-tile routing resources of the four tiles can be used for routing. Altogether, this allows more compact PU designs than splitting one PU onto several individual PR regions.

In case DynaCORE detects a significant change in traffic profile, for example, that a considerable amount of data flows require DES encryption which cannot be handled anymore by a software instance, but less data require AES encryption, the throughput optimised AES core of Figure 9(a) may be replaced by two smaller cores, one for AES and one for

DES. This is illustrated in Figure 9(c). The four tiles are grouped into two independent PR regions consisting of two tiles, respectively. As in the initial scenario only one link for connecting one PU mapped onto the four tiles is provided, the communication infrastructure has to be adapted when changing the PU composition as well. In Figure 9 this is illustrated by replacing the vertical communication tile in the upper right corner with a switch tile.

As mentioned in Section 4.2, the adjustment of PR regions should not be done in the same step as mapping PUs onto the PR region. The required intermediate step is shown in Figure 9(b). Each tile is configured as an individual PR region with bus macros at each border. This minimises the risk of short circuits when reconfiguring one tile with neighbour tiles still holding configuration code with nets originally routed to the tile under reconfiguration. Special precautions also have to be taken for the communication infrastructure when elements have links to the PR regions under reconfiguration. In Figure 9(b), these elements are marked with numbers. As interfering signals arise at links to regions under reconfiguration, the corresponding switch ports have to be disabled. Special NoC messages are used to activate/deactivate switch ports. A detailed description of this mechanism is given in [14]. When switching back from two small PUs to one large PU, the steps shown in Figure 9 have to be done in reverse order.

6. Summary

The presented design technique enables the adjustment of the boundaries and number of PR regions in a system at runtime. No online manipulating of configuration data or adapted versions of design tools are necessary. The implemented system does not require additional FPGA logic resources to provide this feature but the link to the ICAP component. The complexity to provide a system with such features is completely moved to design time. The practicability of the design technique is shown by means of two examples. The first example explains the single steps in detail on a low abstraction level. On application level, the second example demonstrates the advantages a system can achieve when using a system architecture designed for this technique. To conclude with, the proposed technique opens the way for partially reconfigurable designs which use PR modules of varying sizes with standard design tools.

Acknowledgment

This work was funded in part by the German Research Foundation (DFG) within priority programme 1148 under Grant reference Ma 1412/5.

References

[1] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 12–17, Madrid, Spain, August 2006.

[2] "PlanAhead the Fastest Route to Better Design," Data sheet, 2005.

[3] T. Pionteck, C. Albrecht, R. Koch, and E. Maehle, "Adaptive communication architectures for runtime reconfigurable system-on-chips," *Parallel Processing Letters*, vol. 18, no. 2, pp. 275–289, 2008.

[4] F. Dittmann, "Algorithmic skeletons for the programming of reconfigurable systems," in *Proceedings of the 5th IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS '07)*, R. Obermaisser, Y. Nah, P. P. Puschner, and F.-J. Rammig, Eds., vol. 4761 of *Lecture Notes in Computer Science*, pp. 358–367, Springer, Santorini Island, Greece, May 2007.

[5] F. Dittmann, S. Frank, and S. Oberthür, "Algorithmic skeletons for the design of partially reconfigurable systems," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–8, Miami, Fla, USA, April 2008.

[6] M. Cole, *Structured Management of Parallel Computing*, Pitman/The MIT Press, Boston, Mass, USA, 1989.

[7] M. Hübner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs," in *Proceedings of the 13th Workshop on Reconfigurable Architectures (RAW '06)*, pp. 1–8, Rhodes, Greece, April 2006.

[8] C. Schuck, M. Kühnle, M. Hübner, and J. Becker, "A framework for dynamic 2D placement on FPGAs," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–7, Miami, Fla, USA, April 2008.

[9] S. Guccione, D. Levi, and P. Sundararajan, "Bits: Java based interface for reconfigurable computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD '99)*, 1999.

[10] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "A design methodology for communication infrastructures on partially reconfigurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 331–338, Amsterdam, The Netherlands, August 2007.

[11] S. Koh and O. Diessel, "COMMA: a communications methodology for dynamic module reconfiguration in FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 273–274, Napa, Calif, USA, April 2006.

[12] S. Koh and O. Diessel, "Communications infrastructure generation for modular FPGA reconfiguration," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 321–324, Bangkok, Thailand, December 2006.

[13] E. L. Horta, J. W. Lockwood, D. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Proceedings of the 39th Design Automation Conference (DAC '02)*, pp. 343–348, New Orleans, La, USA, June 2002.

[14] T. Pionteck, R. Koch, and C. Albrecht, "Applying partial reconfiguration to networks-on-chips," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 155–160, Madrid, Spain, August 2006.

[15] Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx Application Note 290, Version 1.2, March 2005.

[16] Xilinx, Inc., "Answer Record #25018—Partial Reconfiguration—PlanAhead Flow FAQ/Know Issues for the Early Access," Partial PlanAhead Program, May 2008.
[17] Xilinx, Inc., "Early Access Partial Reconfiguration User Guide," UG208, Version 1.2, September 2008.
[18] C. Albrecht, R. Koch, and T. Pionteck, "On the design of a loosely-coupled run-time reconfigurable network coprocessor," in *Proceedings of the 7th Workshop on Media and Streaming Processors (MSP '05)*, November 2005.