

RUN-TIME PARTIAL RECONFIGURATION FOR REMOVAL, PLACEMENT AND ROUTING ON THE VIRTEX-II PRO

Stefan Raaijmakers and Stephan Wong
Computer Engineering Laboratory,
Delft University of Technology
email: stefan@bugs.nl, J.S.S.M.Wong@ewi.tudelft.nl

ABSTRACT

Reconfigurable computing entails the utilization of a general-purpose processor augmented with a reconfigurable hardware structure (usually an FPGA). Normally, a complete reconfiguration is needed to change the functionality of the FPGA even when the change is minor. Moreover, the complete chip needs to be halted to perform the reconfiguration. Dynamic partial reconfiguration (DPR) provides the possibility to change certain parts of the hardware while other parts of the FPGA remain in use.

In this paper, we propose a solution using dynamic partial reconfiguration which provides a methodology to generate bitstreams for removal of ‘old’ hardware modules, and placement and routing of new hardware modules within an FPGA. Hardware modules may reside at any location and our solution can connect the additional functionality to the remaining running parts of the chip. In addition, bus macros are no longer necessary and we use the Xilinx tools only for generating the modules. We implemented our solution on a Xilinx Virtex-II Pro series FPGA, specifically the XC2VP30 on the XUP board, and demonstrated that the solution is fully functional.

Keywords— Run-Time Partial Reconfiguration, FPGA, Routing, Reconfigurable Computing

1. INTRODUCTION

In the early 60’s, there was an interest to look beyond the conventional general-purpose machines and to develop new computing paradigms. As a result, reconfigurable computing was conceived [1] as a means to extend the capabilities of general-purpose computing. In reconfigurable computing, parts of a program can be described in hardware, which can result in hardware implementations for different stages of execution. With the introduction of FPGAs reconfigurable computing became accessible. Reconfigurable hardware can outperform general-purpose computing for a wide range of algorithms [2]

When reconfiguring an FPGA for a new function, we encounter two problems: First, changing the functionality

of the device suffers from lengthy reconfiguration latencies. Configuring an entire device can take a tenth of a second, up to several seconds. One solution is to *partially* reconfigure the device, replacing only a small portion of the reconfigurable hardware to change a functionality or just some parameters. This reduces the reconfiguration time and can hide reconfiguration latencies during run-time, by reconfiguring a hardware accelerator for later use, while the other accelerators remain active. Second, for each device within a family and each combination of hardware, a new reconfiguration bitstream has to be synthesized. Normally, each device type, even within a family, needs a different reconfiguration bitstream. Current methodologies for generating partial reconfiguration bitstreams entail using an ad-hoc manner for generating full device configuration of each device and module combination, and extracting the differences at compile time [3]. It is a cumbersome method to distribute an application, especially when multiple modules are used and interchanged, and comes with considerable restrictions.

In this paper, we introduce a solution to overcome these problems. This research is focused on developing a method that enables arbitrary removal and placement of hardware implementations, and if needed, perform the necessary routing to disconnect and connect them to other implementations present in the device. It produces full and partial (re-)configuration bitstreams for first-time setup and subsequent transitions. The platform we focus on is the Virtex-II Pro.

This paper is organized as follows. Section 2 presents related work. In Section 3, we discuss the assumptions made and the methodology followed to arrive at our implementation, and the tools we developed for this. Section 4, we give an example to demonstrate how our methodology evolved and to show that it is fully functional. Finally, in Section 5 we draw our conclusions

2. RELATED WORK

Brebner [4] implemented swappable logic units (SLUs) on the XC6200 which were square, fixed sized modules with

standard interfaces at each side. Communication was only possible along these interfaces, no routing is performed. Connecting modules was done by placing them next to each other, both horizontally or vertically. The modules were freely interchangeable, because the underlying FPGA structure was identical.

The Xilinx application note XAPP290 [5] describes two methods for dynamic partial reconfiguration. The modular method divides the FPGA up into portions for specific functions, which must span the full height of the device. No resources inside the module may be shared by external modules, nor may there be any routing passing through. The size of a module is fixed, only modules with a similar shape can be placed inside the reserved space. Communication between modules must be done through bus macros situated at the edges of the module. The difference based method, described as the second method in the application note, is used for small manual changes to the design only.

Bieser, *et al*, [7] use modules that stretch the height of the device. The routing problem is overcome by using a standard shared bus to which the modules attach. This bus makes use of the tristate drivers in the Virtex-II. The designer can merge the module bitstream with the configuration bitstream using an application based on JBits [8]. JBits development has been stopped, there is no support for newer device families like the Virtex-II Pro. Kalte, *et al*, [6] have produced a bitstream manipulation filter in hardware, which can do the same at run-time.

Bobda, *et al*, [9] have proposed two methods. One is based on vertical slots for the modules, which are connected using a reconfigurable multiple bus (RMB). For the second part, they assume an FPGA that is capable of 2d placement and have build a network on chip that can cope with routing packets, even though the structure of the network is irregular.

Sedcole, *et al*, [10] proposed a method, that is more flexible than described in the Xilinx application notes [5]. They provided a way to place hardware cores above each other, whereas the Xilinx method dictates that modules stretch the entire height of the device. The size and positions for these hardware cores have been predetermined. The issue with static routes passing through the modules were resolved by reserving the long-lines as pass-through regions in the modules. Signals are connected to static routing through bus macros. The operations necessary are performed at a bitstream level.

Hübner, *et al* [11] implemented on-line routing using regular routing structures which stretch vertically through the device. A module can be attached to the structure at any location. In this manner, they provide arbitrary placement of modules of any size. The routing structures are lookup table based and have to be reconfigured at the the position where it attaches to the module. The number of signals that pass-

through the structure is limited, due to the use of lookup tables.

We use direct circuit-switched routing, although our technique could be used for all types of communication. The advantages of our technique are mainly due to the routing. An existing configuration can be reused, and new routes can pass-through existing structures on the FPGA. Because there is no use of bus macros, lookup tables, or other techniques for affixing the routing to a predefined position the delay of the signals through the wires can be less, as they can be shorter. The density of the routing is limited only by the available wires in the FPGA, not by 8 lookup-tables per CLB. As long as there are no conflicts with existing structures, we can arbitrarily place modules anywhere on the device, there is no predefined size or location. The only penalty also stems from the routing process, as it is very computationally intensive.

3. THE RECONFIGURATION PROCESS

Our goal is to demonstrate that it is possible to replace a hardware core with another one by manipulating only the bitstreams. We perform operations on the bitstreams only because we want to avoid lengthy synthesis cycles. To achieve this result we take the following steps:

- we propose an approach to replace one hardware module for another
- we limit our work to the resources and wiring which are of most interest
- the bitstream is decomposed for routing and the resources of interest
- tools are made for manipulating the bitstream with the information obtained
- we develop a tool to combine all operations needed

We propose a more idealized method of partial reconfiguration. Replacing a hardware core for another one during run-time is depicted in Figure 1. For reconfiguration, we first need to extract the current configuration of the FPGA from the device, or use a stored copy. Second, we have to identify the hardware core and subtract it from the bitstream. This can be done by specifying an area in which it is known to reside and use an isolation tool to identify the hardware core and its internal wiring, but we prefer to use a stored version.

Third, in the designated area we remove the wiring connecting to external modules, without removing pass-through routing. The extracted, or a stored list of signals is used to unroute the connectivity between the remaining configuration and the module which has been removed. In step

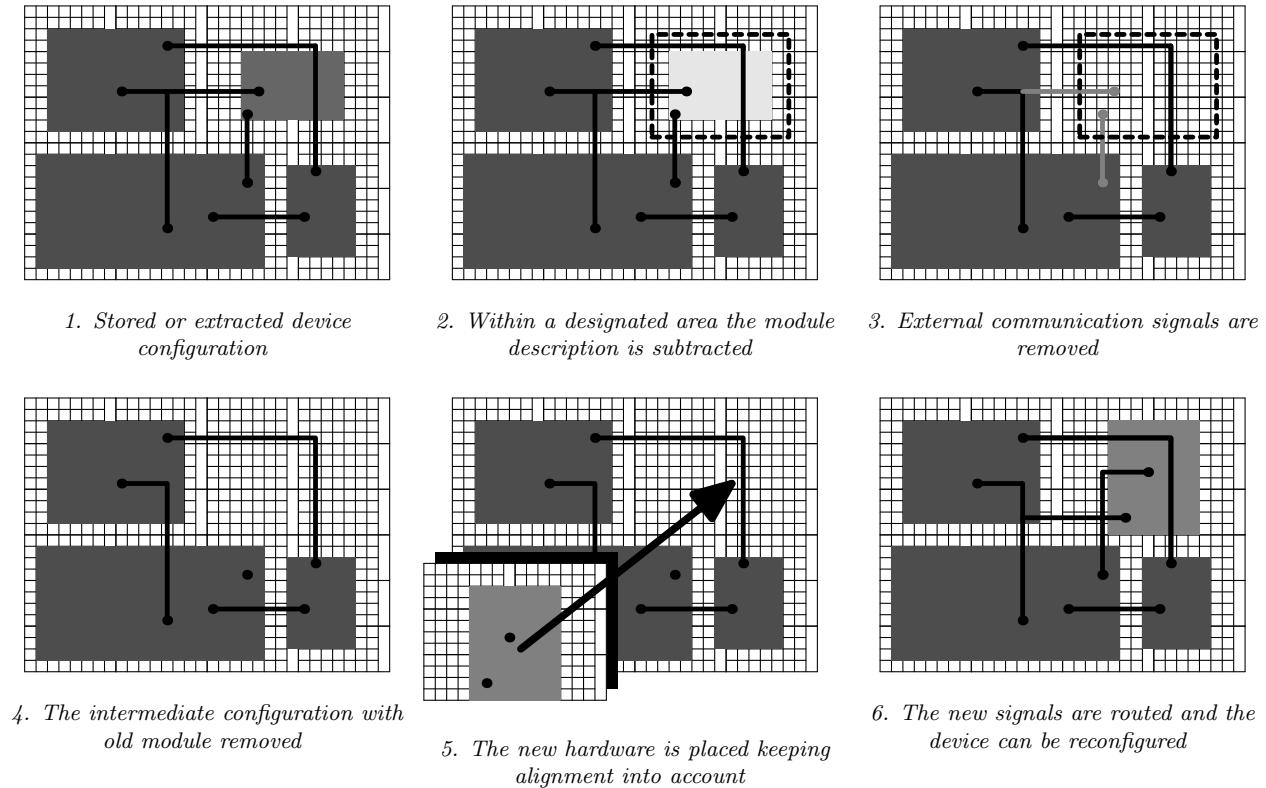


Fig. 1. Exchanging one hardware core for another

four, to ensure there are no damaging effects when directly switching from one core to another, it is advisable to first use this bitstream to remove the core before configuring the new one.

The bitstream for the new core has been previously isolated in a separate bit-file. When placing the new core as depicted in step five it is important that it is aligned properly. The matrix of CLBs is interrupted by a column of BlockRAMs and multipliers each 6 columns. These have a height equivalent to 4 CLBs. Placement may have to be done by steps of 4 vertically and steps of 7 horizontally, depending on the module. The last step involves routing the signals to and from the core. These do not have to be in the same position as the previous core, nor do they have to connect to the same terminals of the hardware remaining in use. With the aid of a netlist the routing is generated and added to the bitstream. The resulting bitstream is the complete description of the new device configuration. This is compared to the original configuration and the intermediate configuration where the old core is removed, to produce the partial bitstream.

Because of the huge number of structures in an FPGA, limitations have to be set to only determine the resources which are most important. For now, our research has been

limited to the use of CLBs. The ISE [12] tools have been used to synthesize the hardware modules. Most effort has been put into the wiring. There are special tri-state bus structures, and long lines, which span the width or height of the entire device. To prevent accidental damage to the device we will not use these wires as they can be driven from multiple locations.

Although Xilinx does provide some information on the bitstreams needed to reconfigure the device, this only covers how the stream is divided into commands and frames. There is no detailed description on which bits to set to provide a pathway for a signal. To obtain this information, we developed a way to build low level device configurations and relate this to the bitstream using the *fpga_editor*. We can derive all the wires which the signals can possibly take through inspection. We obtained enough information to build ad-hoc scripts that systematically generate pathways and produce the bitstreams. We have determined which bits are responsible for selecting a specific route, which was stored in a wire database. In hindsight, it would have been much better to use “*xdl -report -pips*” to determine the device structures, as this method would be a lot less device dependent, or labor intensive. Up till now, we can configure 78% of the pip settings for the XC2VP30. Using the information from *xdl*

we expect to get a complete database of all the configuration bits regarding all resources.

Using the information provided by Xilinx in the user guide [13] for the device family, we have constructed a program that can read the binary bit-file used for configuring the device and translate it to and from a more human readable form, displaying the commands and showing the frame content in hexadecimal notation, and information relating frame contents to columns in the device. As all our tools are written in *perl*, we built tools that perform these manipulations on the textual representation of the bitstream. Adding a new signal path to the bitstream, involves taking their corresponding bits from the wire-database and use the tools to add the bits to an existing bitstream. Similarly, the wire database can be searched for the bits encountered in the bitstream, thereby extracting the switch settings used for a signal path, which can be subtracted if no longer necessary.

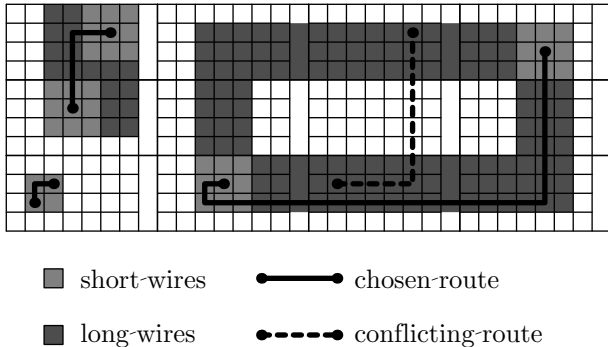


Fig. 2. A simple implementation for routing 2 points

It is cumbersome to do routing by hand and impossible for large networks. We developed a simple implementation of a router which is able to generate a path between two points. The employed algorithm as depicted in Figure 2 searches for all possible routes between the two points. As this is incredibly inefficient, the search space has been limited. We differentiate between short lines connecting only to neighboring CLBs and longer lines connecting CLBs further apart. For short distances, we only search the short lines in the surrounding area. For larger distances, we only allow usage of short wires near the points we want to route between and longer lines for the rest of the pathways, excluding the area in the middle. We also employ a restriction on the number of switches used relative to the distance traveled from the source and we have an absolute maximum to the number of switches. The parameters can be tweaked if no suitable route has been found. The resulting router is far from optimal and very slow, but as building good routers is in a field of its own, for now we are not interested in improving it.

For isolating the various modules from an existing design, we have developed a tool that takes a bitstream and the

region the modules resides in. It extracts just the configuration data of the module and a list of nodes through which it communicates externally. A placement tool relocates the configuration data to another location on the chip. It performs all the necessary checks to test if placement is possible. An unrouting tool takes a device configuration and a netlist containing the source and destination(s) of the routes that have to be removed. This can be a subset of the destinations a particular net has, it only produces a list of switch settings which have to be removed for those particular pathways.

All manipulations are combined in a single tool. A file format has been specified in which the user can assign multiple instances of various modules to user-specified locations. It also has a netlist describing the signals between various modules. A default configuration bitstream and signal positions are used as a base configuration. The first time a composition file is processed, the tool will produce a bitstream fully configuring the device. Consequent runs of the composition file will result in a partial bitstream, first removing the descriptions of the parts no longer present in the new description and then reconfiguring the new functionality.

During the whole process, proof of correctness is difficult. As the scripting facilities of the Xilinx tools were used for gathering data, an error or lack of output while processing these scripts was an indication that there was an error in the data already obtained. Because we can translate the generated pathways to bits in the bitstream, or to a script for *fpga_editor*, we can test whether these produce identical bitstreams. As long as both methods produce the same results we can be confident that our bitstream is correct. As there is no specific knowledge of the internal structures of the FPGA we can only derive some conclusions from the bitstream itself. Inspecting the bits for configuring the switch matrix reveals that each bit only has involvement with driving a specific wire. Because of this, we feel confident that a switch setting for driving one wire will not have a direct influence on the switch settings for another wire, they seem to be independent.

4. AUTOMATING THE PROCESS, USING A COMPOSITION FILE

The composition file is depicted in figure 3. It specifies a pregenerated 'empty' configuration bitstream (in this case configuring the IOBs to take the signals from the switches and drive the LEDs properly), and a file to indicate the location or pads of the external signals available on the chip. Default modules have been created using our isolation tool for the 3-input AND and XOR gates. These were selected because we can not yet relate specific wires to signal names used by the module. The actual composition is described in the definition of instances for a specific module, with a user

```

.cache=./cache
.mods=./mods
.basis_strm=./basis_strm.txt
.basis_nodes=./basis_nodes.txt
.reconfigstrm=./update.txt

.define 3and.txt and1(46,16)
.define 3xor.txt xor1(47,16)

*   in in in out
and1 sw0 sw1 sw2 led0
xor1 sw0 sw1 sw2 -

```

Fig. 3. Device composition input file

specified X-Y location, and a netlist for connecting these instances. There is a special token: '-', for describing an unconnected signal. The first time the build script is run, it produces a full configuration bitstream. The device can be programmed using this bitstream and the *impact* program found in ISE. If a modification is done to the composition file, a partial reconfiguration bitstream is calculated, resulting in a modification to the new configuration. As a series of tests, the 3-input AND is placed at a random location, it is replaced by a XOR at another location. Finally, the AND port is placed again and the output of the AND function is used as an input to the XOR function. All these transitions are functional as expected, but calculating the bitstreams took 225, 229 and 30 seconds respectively on a 2Ghz Pentium M. The area utilization is one lookup table for each module. Bus macros would add two lookup tables for each connection.

5. CONCLUSIONS

Our methods showed that we can remove a piece of hardware configuration on an Virtex-II Pro FPGA and replace it by another. This involves removal of the configuration data and the routing of the old hardware core, after which a new one is placed and connected. These manipulations are done at a bitstream level without the use of the Xilinx tools. We have tested our methods with simplistic hardware to show that it works on a functional level. We have developed tools to automate the process for developing the bitstreams used in run-time partial reconfiguration.

The advantage of our solution is we no longer reserve a specific area for the modules, nor do we make use of reserved routing paths. This means we can arbitrarily place and connect any module to any other module. We do not make use of bus macros, nor do we place any restriction on the size and shape of the modules, although they do have to fit on to the device without causing conflicts with the existing configuration. As long as the router can generate a path-

way, routing can go through existing structures. As long as existing pathways do not conflict with the routing within a new module, modules can be placed on top of existing routing. We can do 2d placement, enabling more efficient area usage and routing densities are only limited by the available resources on the FPGA.

6. REFERENCES

- [1] G. Estrin, "Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3–9, 2002.
- [2] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: the Coming of Age," in *IEEE Trans. on VLSI, Vol. 4, No. 1*, 1996, pp. 56–69.
- [3] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Runtime Reconfiguration," in *DAC '02: Proceedings of the 39th conference on Design automation*. ACM Press, 2002, pp. 343–348.
- [4] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," *fccm*, vol. 00, p. 77, 1997.
- [5] Xilinx, "Two Flows for Partial Reconfiguration: Module Based of Difference Based," <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>, 2004.
- [6] H. Kalte, G. Lee, M. Pormann, and U. Ruckert, "Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems," *ipdps*, vol. 04, p. 151b, 2005.
- [7] C. Bieser, M. Bahlinger, M. Heinz, C. Stops, and K. D. Mueller-Glaser, "A Novel Partial Bitstream Merging Methodology Accelerating Xilinx Vitex-II FPGA Based RP System Setup," in *FPL*. IEEE Circuits and Systems Society, 2006, pp. 701–704.
- [8] Xilinx, "Xilinx JBITS," <http://www.xilinx.com/products/jbits/>.
- [9] C. Bobda and A. Ahmadinia, "Dynamic interconnection of reconfigurable modules on reconfigurable devices," *IEEE Design and Testing*, vol. 22, no. 5, pp. 443–451, 2005.
- [10] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," *Computers and Digital Techniques, IEEE Proceedings*, vol. 153, no. 3, pp. 157–164, 2006.
- [11] M. Hubner, C. Schuck, M. Kuhnle, and J. Becker, "New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Microelectronic Circuits," in *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. IEEE Computer Society, 2006, p. 97.
- [12] Xilinx, "Xilinx ise," http://www.xilinx.com/products/design-resources/design_tool/.
- [13] —, "Virtex II Pro and Virtex II X FPGA User Guide," <http://direct.xilinx.com/bvdocs/userguides/ug012.pdf>, 2004.