

Run-time Placement and Routing on the Virtex 2 Pro

Stefan Raaijmakers and Stephan Wong
Computer Engineering Laboratory,
Delft University of Technology
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 only minor. Moreover, the complete chip needs to be halted to perform the reconfiguration. Dynamic partial reconfiguration (DPR) enables the possibility to change parts of the hardware while other parts of the FPGA remain in use. Nowadays, there are two approaches to accomplish DPR. First, (small) differences between different implementations can be recorded and the functionality of the chip can be subsequently changed by simply transmitting these differences. The main disadvantage is that this approach only works for small differences in functionalities between implementations. Second, the chip can be divided into modules in which (by design) the functionality of the chip can be changed by only changing the modules. The main disadvantage is that the communication between modules must be through dedicated and fixed bus macros and the position of the modules is predetermined.

In this paper, we propose an additional solution to dynamic partial reconfiguration by providing a methodology to generate routing between two arbitrary points within an FPGA. This means that additional functionality can be added to the FPGA at any location and our solution can connect the additional functionality to the already running parts of the chip. In addition, the bus macros are no longer necessary and no synthesis is needed to implement the routing. We implemented our solution on a Xilinx Virtex-2 Pro series FPGA, specifically the XC2VP30 on the XUP board, and demonstrated that the solution works.

Keywords— Reconfigurable Computing, Partial Reconfiguration, FPGA, Routing

I. 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, (part of) a program can be described in hardware, which can result in several hardware implementations for different stages of execution of a single program. Consequently, during program execution the processor has to reconfigure the hardware to the needed functionality. However, the technology in the 60's was not mature enough to sufficiently implement the concept of reconfigurable computing. With the introduction of field-programmable gate arrays (FPGAs) in the early 90's, reconfigurable computing became accessible. Without much doubt, the utilization of reconfigurable hardware adds flexibility without sacrificing much performance, mainly due to the exploitation of parallelism in hardware. This has been proven as the reconfigurable hardware can outperform general-purpose computing for a wide range of algorithms and in some cases by a large margin [2]. It is therefore desirable to attempt to exploit this advantage to accelerate general-purpose computing. Currently, this is done in an ad-hoc fashion, where specifically designed hardware cores are implemented for

each application. This methodology is very much dependent on the platform it targets and is impractical for most software developers.

When reconfiguring a device for a new function, we encounter two problems:

- Changing functionality of the device suffers from lengthy reconfiguration latencies. Delays caused by switching hardware functionality can severely hinder computation. One of the major problems with using FPGAs in reconfigurable computing is the *reconfiguration time*. The current common FPGAs need 1-30 Mbit of configuration data and the fastest configuration method is through an 8-bit interface at 66 MHz [3]. Configuring an entire device can take a tenth of a second to several seconds. One solution is to partially reconfigure the device, replacing only a small portion of the reconfiguration data to change some functionality or parameters. This reduces the reconfiguration time and can increase functional densities for some applications [4]. The granularity of partial reconfiguration varies. The Atmel 94k [5] devices support byte-sized reconfiguration, while the Xilinx Virtex II pro devices support frame sized (1472-9792 bits, depending on the model) reconfiguration. This means Atmel devices have smaller reconfiguration vectors to change the functionality of the device. The Virtex-IV [6] addresses this issue by splitting up the frames in smaller regions.

- For each device within a family and each combination of hardware a new reconfiguration vector has to be synthesized. Normally, each device type, even within a family, needs a different reconfiguration bit stream. For placement of a hardware core, new configuration data for the entire device has to be synthesized. Current methodologies for generating partial reconfiguration entail extracting the difference between the old and new configurations. This means the placement and routing of a coprocessor has to be done at compile time [7]. This is a cumbersome method to distribute an application, especially when multiple cores are used and interchanged.

In this paper, we introduce a solution to overcome these problems. This research is focused on developing a method that enables (semi-)arbitrary placement of hardware implementations, and if needed, perform the necessary routing to connect them to other implementations present in the device. The piece of reconfiguration data can be some sort of hardware core implementing an algorithm, or just a piece of this structure. The resulting reconfiguration vector contains only the difference between the old and new configuration. It removes the resources that are no longer needed and sets the device up for its new functionality. Therefore,

it has a small reconfiguration vector.

The main goal of our experiments is to simply demonstrate the feasibility of our methods. The results are based on very simple hardware descriptions that connect the external switches and leds on an evaluation board to simple hardware structures like functional gates. Our experiments showed that we can arbitrarily place and replace the components at different positions on the FPGA and route the signals.

This paper is organized as follows. Section II presents related work. In Section III, we discuss the assumptions made and the methodology followed to arrive at our implementation. Section IV describes our developed tools, which are needed to perform operations on the bit stream. In Section V, we give an example to demonstrate how our methodology works. Finally, in Section VI we draw some conclusions based on our experiments.

II. RELATED WORK

Sedcole *et al* [8] proposed a method for reconfiguring hardware cores, in the form of modules, that is more flexible than described in the Xilinx application notes [9]. 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 positions for these hardware cores have been predetermined. The issue with static routes through the modules, were resolved by reserving pass through regions in the modules. Signals are connected to static routing through bus macros. The operations necessary are performed at a bit stream level with the use of stored configuration data of the modules.

Hübner *et al* [10] implemented online routing using routing primitives which stretch vertically through the device. A module can be attached to the primitive at any location. In this manner, they provide arbitrary 2D placement of modules of any size. The routing primitives are lookup table based and have to be reconfigured at the the position it attaches to the module. The number of signals that pass through the primitive is limited. The operations are performed by a C programming running on the PowerPC or Microblaze processor using stored configuration data for the routing primitives and modules.

Both methods rely on fixed interfaces between the modules and the routing paths, in the form of a bus macro or lookup table based primitives. We propose a method which does true routing of the signals. This provides more flexibility, as there are no predetermined routing paths or dedicated wires. It makes it possible to route through existing structures even though they do not have specifically reserved space for this. The routing does not make use of lookup tables as with the primitives. This saves resources and reduces delays. Furthermore, there is no limit to the number of signals that can be connected. As all operations are done on a bit stream level, they can be implemented on the PowerPC or Microblaze. We make use of stored configuration data for the module and have a database of configuration data for setting the switches for routing the signals.

III. ASSUMPTIONS AND METHODOLOGY

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

- A device to use for our approach is selected
- The structure and composition of it's resources are reviewed
- We limit our work to the resources and wiring which are of most interest
- The bit stream is decomposed for routing and the resources of interest
- Tools are made for manipulating the bit stream with the information obtained
- We develop a method to use the tools for our goal

Because reconfiguration vectors are specific for a device family, we prefer to chose an accessible common FPGA to work on. Therefore we choose to target the Xilinx Virtex II Pro family. It is in common use, can do partial reconfiguration and the Xilinx university program board [11] has an attractive price tag. It has an xc2vp30 as its main device, offering 2 power PC cores 30k+ logic cells 136 multipliers and 428kBit of ram.

To describe the structures of this series of FPGAs, we use Figure 1 which depicts the arrangement of resources in the XC2VP2, scaled to the number of bits they use in the bit stream. This is the only device in the Virtex II Pro family that does not have a PowerPC core. The most common resource is the complex logic block (CLB). These logic blocks contain a portion for the hardware description and use a switch matrix for routing signals to other resources. Resources like BlockRAMs and RocketIO interfaces are distributed throughout the matrix column-wise at regular intervals. These are connected by BlockRAM interconnections (BRI) and RocketIO interconnections (RII), which are very similar to the switch matrix of the CLB. BlockRAM Interconnections also provide connectivity to the multipliers. Around the edges of the device the connection to the outside world is made through IO interconnections (IOI).

There are different types of wires connecting the various switch matrices. There are wires connecting to the neighboring resources, wires that connect to the resources which are spaced two and four positions apart and wires that connect to resources spaced six and twelve positions apart. The wires run in horizontal and vertical directions. There are also long lines, which span the width or height of the entire device, and special tristate wires. On a device-wide level, there are special wires for distributing clock and global signals throughout the entire device.

For now, we will limit ourselves to the use of CLBs only. The method can be expanded to take the use of the other resources into account. Most of the wires have only a single driver ensuring signals can only travel in one direction. To prevent accidental damage to the device we will not use wires that can be driven from multiple locations. This

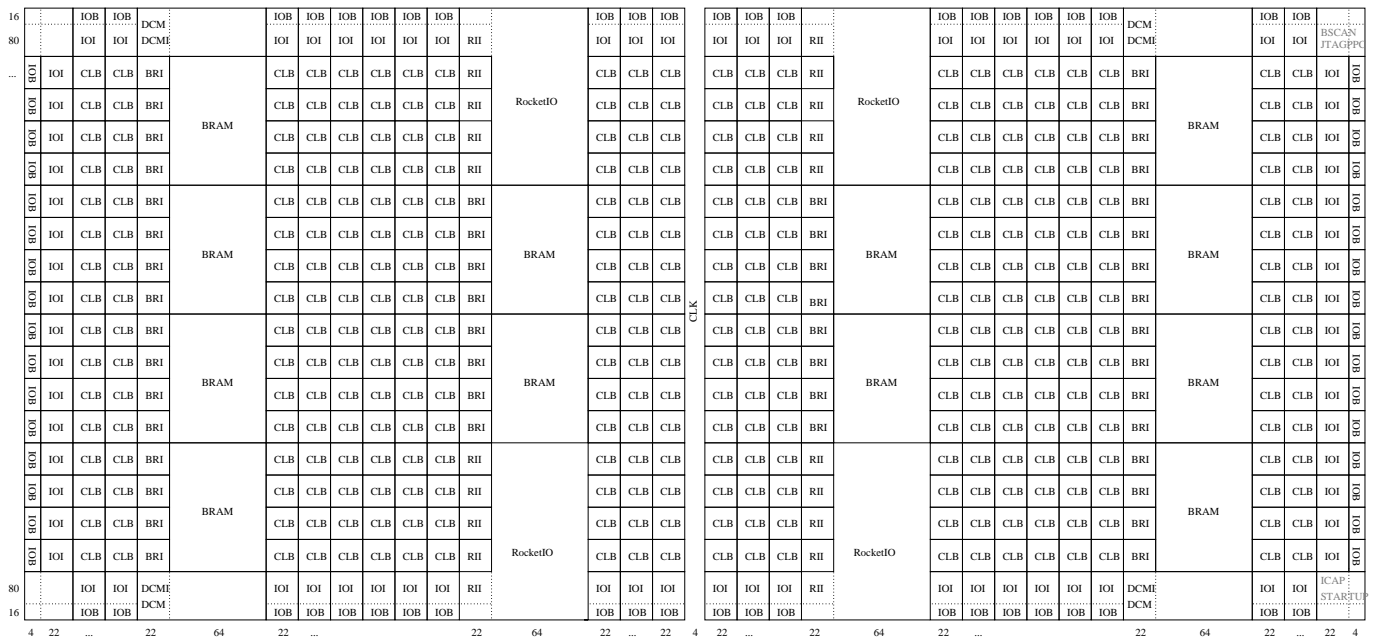


Fig. 1. XC2VP2 resource overview scaled to represent the size of their reconfiguration data in the bit stream

means we exclude the utilization of long lines and tristate lines.

Although Xilinx does provide some information on the bit streams 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. We therefore need a way to build low level device configurations and relate this to the bit stream. The `fpga_editor` program in the ISE [12] software package allows for manual manipulation of the FPGA resources. It provides a visual interface depicting the internals of the FPGA. It also has a scripting facility to store and repeat common manipulations. Among it's functionalities there is a way to specify the wires over which a signal has to be routed. We can use the routing mechanism and can derive all the wires which the signals can possibly take through inspection. This provides enough information to build ad-hoc scripts that systematically generates pathways and produce the bit stream using the `fpga_editor`. From the bit stream, we have determined which bits are responsible for selecting a specific route. This information is stored in a wire database.

With this information we built a tool chain that facilitates the manipulation of bit streams. As the tools are meant for an academic environment, the bit streams are translated to a format better suited for viewing and manual manipulation. The tools can handle this text-based readout after which the bit stream is translated back into a binary format that can be used to program the device.

IV. SOFTWARE TOOLS FRAMEWORK

The development of the tools started with simple bit stream manipulations in the form of adding and subtracting bit-patterns, but these have progressed into more com-

plicated tools. The resulting tool chain now enables the extraction and addition of hardware to a running configuration. These operations are done on a bit stream level, ensuring that the developed techniques can be translated in a method for partially reconfiguring hardware on demand during runtime.

```
Source file: top.ncd, part: 2vp30ff896, date: 2006/04/12, \
time: 11:17:46, length: 82100 bytes
30008001 Type 1 packet, write 1 word(s) to CMD : RCRC
3001c001 Type 1 packet, write 1 word(s) to IDCODE 0127e093
30012001 Type 1 packet, write 1 word(s) to CDR 00053fe5
30008001 Type 1 packet, write 1 word(s) to CMD : SHUTDOWN
30000001 Type 1 packet, write 1 word(s) to CRC 00007a94
20000000 Type 1 packet, 0 word(s) to CRC
20000000 Type 1 packet, 0 word(s) to CRC
20000000 Type 1 packet, 0 word(s) to CRC
20000000 Type 1 packet, 0 word(s) to CRC
30008001 Type 1 packet, write 1 word(s) to CMD : AGHIGH
30008001 Type 1 packet, write 1 word(s) to CMD : WCFG
30002001 Type 1 packet, write 1 word(s) to FAR \
ba:0 mja:3 mna:4 byte:0 -> CLB 1
300040ce Type 1 packet, write 206 word(s) to FDRI
0,3,4 IOB : 0300 *****
0,3,4 IOI 082: 0000 0000 0000 0000 0000
0,3,4 CLB 081: 0000 0000 0000 0000 0000
0,3,4 CLB 080: 0000 0000 0000 0000 0000
0,3,4 CLB 079: 0000 0000 0000 0000 0000
0,3,4 CLB 078: 0000 0000 0000 0000 0000
0,3,4 CLB 077: 0000 0000 0000 0000 0000
0,3,4 CLB 076: 0000 0000 0000 0000 0000
0,3,4 CLB 075: 0000 0000 0000 0000 0000
0,3,4 CLB 074: 0000 0000 0000 0000 0000
0,3,4 CLB 073: 0000 0000 0000 0000 0000
0,3,4 CLB 072: 0000 0000 0000 0000 0000
0,3,4 CLB 071: 0000 0000 0000 0000 0000
0,3,4 CLB 070: 0000 0000 0000 0000 0000
...
```

Fig. 2. Text representation of a bit file partially reconfiguring the 5th frame of the first column

Using the information provided by Xilinx in the user

guide [3] for the device family, the binary bit file used for configuring the device is translated in a more human readable form, displaying the commands and showing the frames in hexadecimal notation. The data sheet also provides the information to show which frame relates to which column in the device. As the number CLBs in a row is known and they are distributed evenly over the device it can be deduced to which row the bits in a frames are related. Each line of frame data in the text file therefore contains the equivalent of 80 bits, which are all the bits related to a CLB in a certain row. The data is preceded by the frame address, the type of data, and the row number. The output showing the first 28 lines of a bit file used for partially reconfiguring a xc2vp30 is depicted in Figure 2. There is also a tool translating this human readable form back to a bit file which can be used to program the device.

Replacing the bits of a certain CLB therefore becomes equivalent to replacing certain lines in the textual representation of the bit file. Addition or subtraction of bits from the bit stream for a certain CLB means adding or subtracting these bits from a few lines of the text file. The tools for adding, subtracting, and replacement of bits determine for the frame and row addresses at the beginning of the line and perform the operation on the data in that line.

The snippets of bit streams stored in the wire database are represented in a similar format. Adding a new signal path to the bit stream is the equivalent of prefixing the appropriate address for frame and row to the lines in the wire database and use the tools to add the bits to an existing bit stream. Similarly, the wire database can be searched for the bits encountered in the bit stream, thereby extracting the switch settings used for a signal path. With these switch settings the routing of a signal in the current configuration can be retrieved.

With the same information we can also route a signal between two points. It is fairly cumbersome to do routing by hand as is the case with the `fpga_editor` tool. We have developed a simple implementation of a router which is able to generate a path between two points. The employed algorithm as depicted in Figure 3 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 spaced 2 or 6 positions from another. 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. The search space for long lines is restricted to the rectangular area between the points, 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 generated pathways are sorted to length and compared with the bit file for conflicting use of wires. The paths generated are steel square shaped. The parameters can be tweaked if no suitable route has been found.

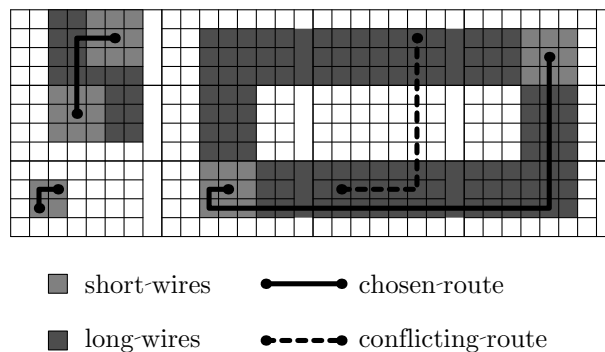


Fig. 3. A simple implementation for routing 2 points

V. REPLACING HARDWARE CORES

Replacing a hardware core for another one is depicted in Figure 4. For the reconfiguration we require a description of the current configuration, an isolated configuration file of the new core, as well as a net list for the signals that have to be routed between the new core and the existing design. For reconfiguring the core we need to first extract the current configuration of the FPGA from the device or use a copy of the configuration file it was configured with. We have to identify the hardware core which has to be removed from the bit stream. This can be done by specifying a specific area in which it is known the core resides. It is also possible to search for a specific core in the bit stream by comparing it with the stored version. For now we assume it is known where the core resides.

In the designated area we isolate the hardware core and we can trace internal wires and external signals. As we only make use of the lines that have a single driver the signals can only be incoming or outgoing. We have to take into account that incoming signals can also be in use elsewhere in the device, so we can not eliminate the entire net. Outgoing signals can be eliminated entirely. The bits that have been found to be a part of the hardware core or its internal wiring can be subtracted from the bit stream. To make sure there are no damaging effects when directly switching from one core to another, it is advisable to first use this bit stream to remove the core before configuring the new one.

The bit stream for the new core has been previously isolated in a separate bit file. When placing the new core it is important that it is aligned properly. The matrix of CLBs is interrupted by a column of BlockRAMs and multipliers at an interval of 6 columns. These have a height equivalent of 4 CLBs. If BlockRAM or CLB resources are used by the core it is important that positioning it in the vertical direction is done in steps of 4. Most of the time horizontal displacement can only be done with multiples of 7 because of the column of BlockRAMs and multipliers. 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 net list the routing is automatically generated and added to the bit stream. The resulting bit stream is the complete

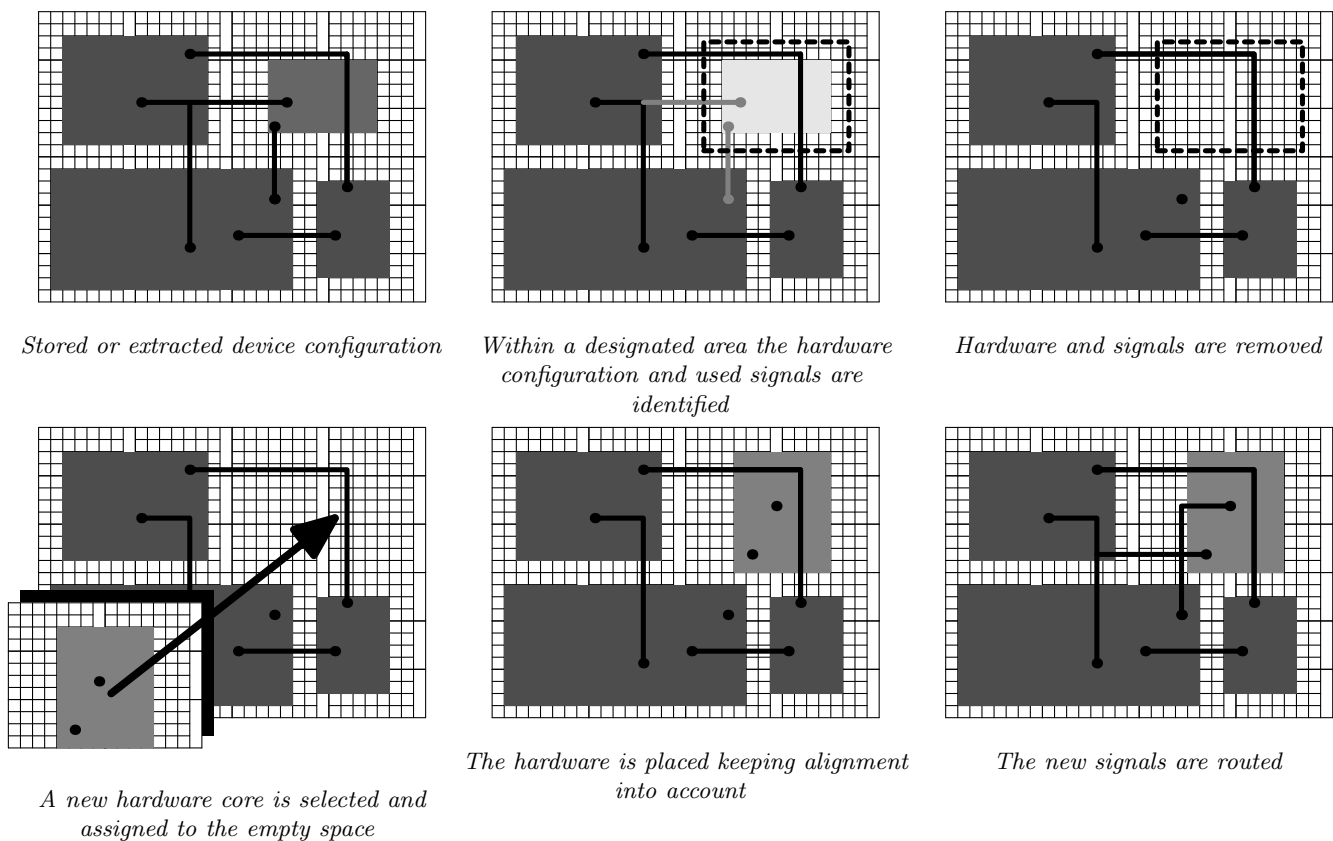


Fig. 4. Exchanging one hardware core for another

description of the new device configuration. This is compared with the original configuration and only the frames that are different will be reconfigured.

VI. CONCLUSIONS

Our methods showed that we can remove a hardware description from an Virtex II Pro FPGA and replace it by another. This involves removal of the configuration data and the routing of a hardware core, after which a new one is placed and connected. These manipulations are done at a bit stream 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. Although some manipulations on the bit stream are performed manually we have build a primitive router for connecting two points in the FPGA.

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*, pp. 56–69, 1996.
- [3] Xilinx, "Virtex II Pro and Virtex II X FPGA User Guide." <http://direct.xilinx.com/bvdocs/userguides/ug012.pdf>, 2004.
- [4] J. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems," in *IEEE Symposium on FPGAs for Custom Computing Machines* (P. Athanas and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 78–84, IEEE Computer Society Press, 1995.
- [5] Atmel, "AT94KAL Series Field Programmable System Level Integrated Circuit." http://www.atmel.com/dyn/resources/prod_documents/doc1138.pdf, 2004.
- [6] Xilinx, "Virtex 4 Configuration Guide." <http://direct.xilinx.com/bvdocs/userguides/ug071.pdf>, 2004.
- [7] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," in *DAC '02: Proceedings of the 39th conference on Design automation*, (New York, NY, USA), pp. 343–348, ACM Press, 2002.
- [8] N. P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular Partial Reconfiguration in Virtex FPGAs," in *FPL*, pp. 211–216, 2005.
- [9] Xilinx, "Two Flows for Partial Reconfiguration: Module Based of Difference Based." www.xilinx.com/bvdocs/appnotes/xapp290.pdf, 2004.
- [10] 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*, (Washington, DC, USA), p. 97, IEEE Computer Society, 2006.
- [11] "The Xilinx XUP-V2Pro Board." <http://www.xilinx.com/univ/xupv2p.html>.
- [12] "Xilinx ISE." http://www.xilinx.com/products/design_resources/design_tool/.