

Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration

Norbert Abel, Udo Kebschull,

Kirchhoff Institute for Physics, Heidelberg University
INF 227, 69120 Heidelberg abel@kip.uni-heidelberg.de

Keywords. FPGA, DPR, HLS, Object-Orientation

1 Introduction

Nowadays, two innovative future trends regarding hardware development and hardware description can be found. The first trend concerns the hardware itself. Modern Xilinx FPGAs provide the possibility to be reconfigured partially and dynamically – which is called dynamical partial reconfiguration (DPR). DPR means that parts of the hardware can be exchanged while the rest of the circuit is running untouched. This opens a huge field of new functionalities on FPGAs. Examples of applications which can be improved using DPR are video processing, low-power, automotive or packet filtering. Other technologies like dynamically reconfigurable processors or scrubbing are not even possible without DPR. Therefore, DPR is a powerful and promising technology. Unfortunately, it is still very challenging to use. Today, using DPR means struggling with architectural details of the used FPGAs and the according synthesis and implementation tools. A developer would focus most of the time on DPR and only a small part of the time on the implementation of the actual modules – of course that is the opposite of what hardware engineers want to do.

The second trend concerns the way hardware is described. The rapidly increasing complexity of hardware led to the usage of computer aided hardware design and to the development of hardware description languages (HDLs). Today, the most important HDLs are VHDL and Verilog. Although these HDLs already allow to describe hardware on a very high level, the developer still has to handle registers, clocks and clock domains. Using an HDL operating on the algorithmic level, this is not necessary any longer. Here, designs can be described exactly as they are in software languages like C, without the need to care about registers or clocks – which is called high level synthesis (HLS). Many hardware developing groups are looking forward to an HDL which operates on the algorithmic level, since this would come with a significant increase in productivity. The aim is to be able to translate common software algorithms to hardware in an efficient way. In that process, the highest challenge is to make use of the hardware's intrinsic parallelism. [1]

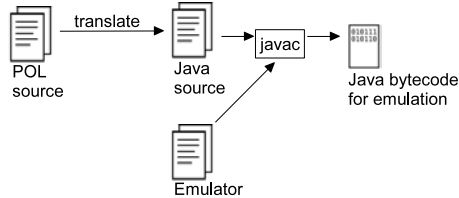


Fig. 1. The emulation toolflow of the DPR framework

Although both DPR and HLS are important future trends regarding hardware design, they develop quite independently. Today’s software-to-hardware compilers focus on conventional hardware and therefore have to remove dynamic aspects such as the instantiation of calculating modules at runtime [2]. Even object-oriented languages like SystemC [3] do not support the dynamic instantiation of objects (that means the usage of *new* or *delete* outside of the constructor) for synthesis at all [4]. On the other hand, DPR tools are working on the lowest possible layer regarding FPGAs: the bitfile level. This paper focuses on the design and the implementation of a Framework combining the two technologies, since this has the potential to kill two birds with one stone. Firstly, DPR can change the programming paradigm in future HDLs regarding dynamic instantiations. Dynamic parts would not have to be removed any longer but could be realized on the target FPGA using DPR. Secondly, a high-level language support of DPR technologies could help end its shadowy existence and turn it into a commonly used method.

2 The Framework

Our DPR Framework consists of a software-to-hardware compiler, an NoC also reliable for data buffering, a Merger, an adaptive Scheduler and a Java Emulator. Figure 1 and 2 illustrate how these components interact with each other and with the Xilinx DPR tools. The reconfigurable modules are described in a Java-like language called POL (Parallel Object Language). For testing and verification, the POL sources can be translated to Java. The Java Emulator is then able to execute these Java files, giving a developer a first and quick chance to verify the correctness of the program. The POL sources can also be translated to VHDL. The generated VHDL files are plugged into a Network on Chip called CM (Communication Matrix). The CM is responsible for the data forwarding between the modules and for the data buffering. The Merger combines the generated VHDL files and the CM files and starts several XST (Xilinx Synthesis Tool) synthesis runs, until all required netlist files (called NGC files) are created. These files represent the functionality of the static area and the functionality of the reconfigurable modules. They are used as input files for Xilinx PlanAhead, which eventually does the DPR floorplanning and generates the (partial) bitfiles using the Xilinx ISE Early Access tools. The CM is controlled by a Scheduler

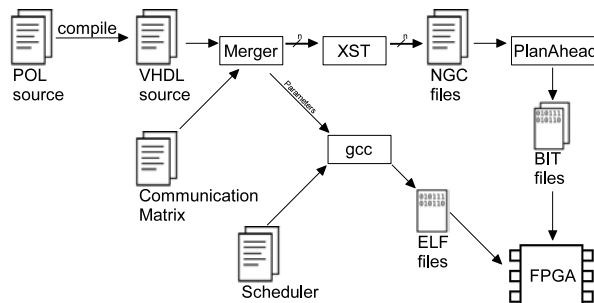


Fig. 2. The implementation toolflow of the DPR framework

running in software on the Virtex' PowerPC. It determines which reconfigurable module is loaded into which PRR (partially reconfigurable region) at which time. Since the Scheduler also runs on the target FPGA, the reconfiguration has to be done via the ICAP (Internal Configuration Access Port [5]). Therefore, the generated design is a self-reconfiguring system.

2.1 POL and Object-Orientation

As a first step, we analyzed the typical structure and behavior of reconfigurable hardware. It turned out that this hardware is comparable in many aspects to object-oriented software [6]. The several hardware elements (like LUTs or flip flops) can be seen as hardware objects, which have a simple state and provide simple methods to the surrounding system. These small hardware objects can be aggregated to bigger objects, like adders or multipliers, which have a more complex state and provide more complex methods. From this point of view, reconfigurable modules can be seen as complex hardware objects. Using DPR, it now becomes possible to create and delete these objects at runtime, just like software objects. Thus, DPR enables a one-to-one correlation between software and hardware objects. This way, our software-to-hardware compiler directly translates software objects (described in POL) to hardware objects (described in VHDL)[7].

Using object-orientation, all objects are declared once, but can be instantiated multiple times. Thus, it is important to distinguish between the classes and their instances. Regarding hardware, the classes are represented by partial bitfiles. An instance is created when the partial bitfile is loaded onto the FPGA (into a PRR). It is obvious that one hardware class can be instantiated many times.

Using POL, the instantiation of a new instance can be done with a simple *new*, but a few design rules have to be considered. The communication between the objects is realized with *signals* and *slots* (inspired by Qt [8]). No public attributes and no direct attribute access are allowed. For inter-object communication the *get()* and *emit()* methods have to be used. The method *calc()* is running permanently and represents the continuous characteristic of hardware modules. It is

the only method containing functionality. Due to the strict restrictions of POL, it is possible to translate the POL objects directly to parallel running hardware components. Every POL *slot* is translated to a VHDL input signal. Every POL *signal* becomes a VHDL output signal and the functionality of the components is extracted from the method *calc()*. The method *connect* tells the CM which VHDL output signal has to be connected to which VHDL input signal via the CM. The method *emit()* sends one data item to the CM, *get()* receives one data item from the CM. In order to be able to implement manually optimized VHDL code, POL provides the *component* statement, which makes it possible to include submodules written directly in VHDL.

2.2 Short Reconfiguration

The object-oriented approach can help to significantly decrease the reconfiguration time. In conventional DPR setups, the switch from one hardware module to another is always correlated with the read-out of the complete PRR, in order to save the current state of the module. Using object-orientation, the change from one instance of a class to another instance of the same class only requires a change of the object's state. The functionality stays the same. Thus, we decided to separate the functionality and the context to be able to reconfigure them individually. For this, the hardware modules have to store their state in a particular BRAM⁴ before swap-out, and to load their state out of the BRAM after swap-in. As a consequence, the change from one instance to another instance of a single class only causes the reconfiguration of the BRAM. This is called Short Reconfiguration. In contrast, the change from an instance of class *A* to an instance of class *B* is called Long Reconfiguration, since the complete PRR has to be overwritten via DPR. Using this new approach, the read-back of the complete PRR can always be omitted, even if a Long Reconfiguration is performed.

2.3 Communication Matrix

In many applications, programmable hardware is used to handle huge data streams. Thus, the dynamic hardware generated by a POL-to-hardware compiler has to be able to handle such huge data streams. This negates the usage of a simple bus to let the hardware modules communicate with each other. Using a bus, the hardware modules would be calculating in parallel but would have to provide the result of their calculation sequentially. Regarding big data streams this would negate the whole parallelism. To avoid this kind of bottleneck, the Communication Matrix provides a parallel inter-object communication, based on a set of FIFOs⁵ and multiplexers. All FIFO elements consist of a target address (the hardware instance the data is sent to) and the payload. Based on the target address and the placement information coming from the scheduler, the

⁴ Block Random Access Memory – built-in RAM blocks of the Virtex chips

⁵ First In First Out buffers

multiplexers can decide which FIFO has to be linked to which PRR. The CM contains one FIFO per hardware class. Multiple instances of one hardware class share a single FIFO. This approach makes it possible to handle the data streams of several hardware objects in parallel, but avoids the instantiation of too many FIFOs. Our current framework supports 16 classes and 64 instances per class. If the matrix instantiated a FIFO for every possible instance, it would have to provide 1024 FIFOs. Using one single FIFO per class, the matrix only has to instantiate 16 FIFOs. Furthermore, the number of classes is well known at the compile time, while in contrast the number of instances cannot be known during compilation. Therefore, the matrix only has to instantiate as many FIFOs as the number of classes that are used [9].

2.4 Emulator

The usage of partial reconfiguration leads to an extended toolflow regarding synthesis, mapping, placement, routing and bitfile generation. All these steps have to be done for the static area, and each reconfigurable module times the number of PRRs. For example a design with 2 PRRs and 3 hardware classes requires 7 runs for each of the steps. Therefore, the implementation time increases significantly. In our test setups, the generation of a completely static design for a V4FX20 required about 30 minutes. The generation of a partial design with 2 PRRs and 3 hardware classes required about 3 hours. It is therefore obvious that a developer needs the possibility to verify his POL code, without the need for implementation. Thus, we created the Java Emulator. It uses the concept of threads in Java to recreate structures and mechanisms (namely the FIFOs and multiplexers) of the Communication Matrix. The Emulator serves as an extra layer between POL code and Java Virtual Machine. Rebuilding all characteristics and features of the Communication Matrix allows the execution of POL in software with exactly the same behavior as in hardware [10].

2.5 Scheduling

In our current setup, the Scheduler is running in software on a PowerPC inside the Virtex chip. It is responsible for loading the correct instance of the correct class based on the data coming out of the buffers. It also manages the creation and the destruction of instances. For this, the PRRs (and therefore the loaded instances) are connected to the scheduler via DCB (Dynamic Control Bus). This enables the hardware instances to request the instantiation or the removal of an instance. Based on the number of items inside a FIFO, the Scheduler decides which class has to be loaded next (via Long Reconfiguration). In that process, it is the task of the Scheduler to keep the number of Long Reconfigurations as small as possible. Since all instances of one class share a single FIFO, the Scheduler has to check continuously which instance is addressed by the top item coming out of the FIFO. This instance has to be loaded via Short Reconfiguration.

3 Case Studies and Measurements

3.1 Dynamic Object Instantiation

Our first example implementation focused on the functionality of the framework itself. To be able to do Runtime Scheduling, instantiating reconfigurable modules dynamically, their alternating execution (swap out, saving the context, swap in, restoring the context) and the final removal of instances have to work correctly. To verify this functionality, we chose the intuitive and playful example Pong. It is a simple game for two players. Each player can move a bar up and down on his side of the screen and tries to bounce the ball back when it arrives on his side. Our implementation of the game is able to contain more than one ball. New balls can be added to the game by simply pressing a button. A ball can also leave the game when it is missed by a player. To verify the framework, each of the balls and the bars have been represented as a reconfigurable module. It is obvious that the creation of a new ball correlates to the instantiation of a new hardware instance of the ball class and the removal of a ball correlates to the removal of a ball instance. This example also demonstrates the power of the object-oriented approach. All balls are implementing the same functionality. Thus, instantiating multiple balls can be done with ease since no new bitfile is required. Furthermore, a switch from one instance of a class to another instance of the same class requires only a Short Reconfiguration.

The system has been instantiated on a Xilinx ML405 board containing a Virtex-4 FX20. For synthesis, mapping, placement, routing and bitfile generation, we used ISE 9.1 SP 2 extended by the partial reconfiguration patch (available via Xilinx' Early Access Page [11]). For PowerPC subsystem and software gener-

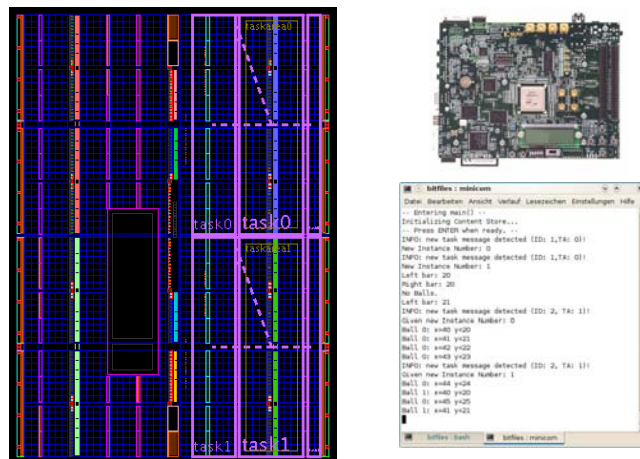


Fig. 3. Left: Screenshot of PlanAhead's floorplan
 Top right: Picture of the ML405 development board
 Bottom right: Output of the Pong example running on the ML405

ation, we used the EDK 9.1 SP 2. The floorplanning and the control of the specific partial reconfiguration placement, routing and bitfile merging was done with PlanAhead 10.1.1. Figure 3 shows a screenshot of PlanAhead’s floorplan. In our example, the size of the PRR was 256 CLBs⁶ (the according partial bitfile had a size of 129 kB). It took 323 microseconds to reconfigure the PRR. The state of the reconfigurable module was stored in 4 BRAMs (the according partial bitfile had a size of 10 kB). Thus, the Short Reconfiguration took 52 microseconds. Saving the state of a reconfigurable module in the BRAM took 300 ns, restoring it took 200 ns. Compared to the reconfiguration times, these values are negligible. Using two PRRs, one for the Balls and one for the Bars, only the Short Reconfiguration had to be performed. This way, the utilization of the Short Reconfiguration leads to a reconfiguration speedup of 10. Our tests proved that the framework translates the object-oriented representation to valid VHDL code. The resulting hardware cooperated nicely with the Communication Matrix. The swap-in / swap-out mechanism worked as expected. The dynamic generation and deletion of new Balls was successful. Furthermore, the framework generated Java code with the same functionality that could be used without modification for testing and evaluation.

3.2 Streaming

In many applications, FPGAs are used to handle data streams with a high throughput. Thus, we evaluated the possible implementation of an audio DSP application, demonstrating the behavior of a dynamic environment regarding data streams, on a Virtex-4 FX40, using the same tools as in the case study above. Our audio DSP application has two 16-bit inputs (stereo) and two 16-bit outputs. We implemented 4 hardware classes representing 4 different effects: a high pass filter, a low pass filter, a distortion and an echo. Every class is instantiated to activate the corresponding effect on a single channel and removed to deactivate it. Due to the 2 channels (left and right), every class has 2 instances at most. The Framework provides 4 PRRs. Thus, if the user activates all effects on both channels, the scheduler uses each PRR for one single hardware class and changes its instances alternately. An uninterrupted audio stream at the outputs is possible since the hardware modules could calculate 100 000 samples in one millisecond (due to the clock frequency of 100 MHz), but only 48 samples per millisecond were needed (due to the sampling rate of the audio stream). The scheduler needs 52 microseconds for Short Reconfiguration. Every reconfigurable module stays configured for 1 000 microseconds. The PRRs cannot be reconfigured simultaneously but only sequentially, since the FPGA’s reconfiguration unit can only reconfigure one frame at once. The Communication Matrix has to store at least $48\,000 \text{ samples/s} \cdot 2\,104 \mu\text{s} \approx 101 \text{ samples}$. Thus, every instance calculates 1 000 microseconds and pauses 1 104 microseconds. In this example, one reconfiguration turn lasts 2 104 microseconds. 2 000 microseconds of a turn are used for calculation. 104 microseconds are used for reconfiguration.

⁶ Configurable Logic Blocks – a cluster of 4 Slices

Every PRR had the size of 812 CLBs. The static area (containing the Communication Matrix) had a size of 1008 CLBs. Therefore, the resulting hardware design would have had a size of 4656 CLBs (that is the complete V4FX40). If one had instantiated all needed instances at once (without using DPR and CM), the design would have had a size of about $812 \cdot 8 = 6496$ CLBs. Thus, DPR helped increase the FPGA's capacity utilization by 28%. This value increases with the number of audio channels. For example, using 8 audio channels would lead to a utilization increase of 82% (4656 CLBs with DPR instead of 25984 CLBs without).

Please note that these results are based on the utilization of the Short Reconfiguration. Without using this technology, a turn would have at least the length of 12 880 microseconds and 3 220 microseconds would have to be used for reconfiguration (two times swap in and swap out). So without the usage of the Short Reconfiguration technology, 25% of a single turn would be used for reconfiguration and the Communication Matrix would have to store 619 samples per instance – whereas utilizing the Short Reconfiguration, only 5% of a turn are used for reconfiguration and the Communication Matrix only has to store 101 samples per instance.

4 Conclusion

The object-oriented DPR Framework enables hardware developers to write reconfigurable modules without going into detail with the reconfiguration techniques. The reconfigurable modules are described in a high-level language called POL. Since POL is very similar to Java, it makes use of well-known programming constructs. It allows expressing the needed parallelism explicitly, without introducing statements which are unusual in software programming (such as *par* in Handel-C [12]). The dynamic instantiation of reconfigurable modules can be done with a simple *new*. This makes the handling of the whole reconfiguration process much easier and helps increase the productivity significantly. Current case studies are very promising and show that the Framework is ready to be used even in high data rate environments. [13]

References

1. S. Edwards, *The Challenges of Hardware Synthesis from C-like Languages*, DATE04, Paris, France, 2004
2. R. Helaihel and K. Olukotun, *Java as a specification language for hardware-software systems*, International Conference on Computer Aided Design, 1997
3. T. Grötter et al., *System Design with SystemC*, Kluwer, 2002
4. <http://www.systemc.org>, *SystemC Synthesizable Subset*, 2009
5. Xilinx, Inc., *Virtex-4 FPGA Configuration User Guide*, 2008
6. N. Abel and U. Keschull, *Parallel Hardware Objects for Dynamically Partial Reconfiguration*, FPL08, Heidelberg, Germany, 2008
7. F. Grüll, *The Parallel Object Language*, Diploma Thesis, Heidelberg, Germany, 2009
8. Qt Homepage, <http://qt.nokia.com/>
9. N. Meier, *Development of a Framework for Dynamic Partial Reconfiguration serving the Object Oriented Hardware Programming Language POL*, Diploma Thesis, Heidelberg, Germany, 2009
10. A. Beyer, *Development of an Emulator for the execution of POL-Code inside a Java-Environment*, Bachelor Thesis, Heidelberg, Germany, 2008
11. Xilinx, Inc., *Early Access Partial Reconfiguration User Guide*, ug208, 2008
12. Agility Design Solutions Inc., *Handel-C Language Reference Manual*, 2010
13. N. Abel, *Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration*, PhD Forum, FPL10, Milano, Italy, 2010