PARALLEL HARDWARE OBJECTS FOR DYNAMICALLY PARTIAL RECONFIGURATION

Norbert Abel, Udo Kebschull

Kirchhoff Institute for Physics,
Heidelberg
INF 227, 69120 Heidelberg
email: {abel, kebschull}@kip.uni-heidelberg.de

ABSTRACT

Many of today's software-to-hardware compiler projects try to find data flow parallelism in a sequential program description and use it to generate parallel running hardware components. In this paper we present a new possibility to do a parallel description based on the combination of object-oriented programming and dynamically partial reconfiguration. Our compiler translates software objects directly to Hardware Objects, which are running in parallel and can be instantiated and removed dynamically. Furthermore, we focus on parallel inter object communication which allows the Hardware Objects to communicate in parallel.

1. INTRODUCTION

Since the 1960s object-oriented programming (OOP) has changed the way software design is done. Using OOP, the central point of software development became interacting objects with assigned attributes and methods. The difference between OOP and procedural programming is crucial, since for procedural programming the handled variables are absolutely passive whereas for OOP every object acts as an active part of the program. This means, that the objects provide the attributes and the functionality [1]. Due to this, we use OOP to realize parallel programming. Combined with the thread concept, the described objects really are independently acting instances. This leads to a powerful process description, making it possible to use the parallelism provided by parallel architectures like modern CPUs or FPGAs.

Xilinx FPGAs provide the possibility to be reconfigured partially and dynamically. This means, that parts of the hardware can be exchanged while the rest of the circuit is running untouched. This provides completely new possibilities regarding object-oriented hardware descriptions. The dynamic character of interacting objects can be implemented directly using dynamically partial reconfiguration (DPR). Objects are translated directly into Hardware Objects which, in a simple case, are adders or multipliers, and in more complex cases are tasks consisting of many functional units. Using DPR these Hardware Objects can be loaded and removed dynamically.

There are several former projects that focus on the design of parallel systems in high level languages. Most of them focus on the analysis of a sequential process description. They try to find independent dataflows to execute them in parallel [2]. It turned out that this method is limited. To generate well parallelizable code, the high level programmers often have to write the programs in a special, parallelization aware way. As a consequence new programming languages for highly parallel architectures (like modern graphic cards) provide the possibility to **explicitly** formulate the parallelism [3]. Our paper focuses the design and the implementation of a hardware compiler that takes an explicitly parallel, object-oriented description and translates it to dynamically reconfigurable hardware. Former projects that focus on OOP and reprogrammable hardware usually confined themself to instantiate coprocessor objects on an FPGA. In these environments the communication always stays controlled by a processor or a bus. The hardware instantiation of objects is only done to speed up the object execution but not to run several objects in parallel [4]. Other projects, like the JHDL project [5], focus on the instantiation of parallel running objects, but leave out the possibility of runtime reconfiguration. Although the JHDL Papers talk about dynamic object instantiation, it has never been implemented within the JHDL project. The target of our project is to fully use the intrinsic parallelism of objects to describe parallel hardware, which is mainly consisting of parallel running, dynamically instantiatable Hardware Objects. In section 2 we take a closer look at the correlation between hardware components and software objects. Section 3 focuses on the dynamic linking of objects. Section 4 goes into detail with the implementation. In section 5 we present example applications and in section 6 we summarize our work.

2. HARDWARE OBJECTS

Our basic concept is to find and to use similarities between the objects in OOP and the programmable hardware. For in-

stance, using Java we would not translate the Java code into Java bytecode and then translate the bytecode into VHDL, but we would rather translate the Java objects directly to Hardware Objects represented in VHDL. The main target of this concept is to use the potential parallelism of OOP.

This section demonstrates the correlation between hardware and Java objects. We decided to use Java and VHDL, since Java is object-oriented and supports the parallelism concept and VHDL is very easy to read and to understand. Nevertheless it is not mandatory to use Java and VHDL – other languages like C++ and Verilog also provide the needed functionality. We do not want to focus on the languages but on the principle behind them.

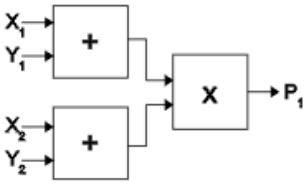

Fig. 1. A simple dataflow

```
//DECLARATION:
class Adder extends HardwareThread {
   private int a, b, s;
   public set_a(int a) { this.a = a; }
   public set_b(int b) { this.b = b; }
   public get_s() { return this.s; }
   protected void calc() { s = a + b; }
}
class Multiplier extends HardwareThread {
   private int a, b, p;
   public set_a(int a) { this.a = a; }
   public set_b(int b) { this.b = b; }
   public get_p() { return this.p; }
   protected void calc() { p = a * b; }
}
public class simpleJava {
  public static void main (String[] args) {
     //INSTANTIATION:
     Adder A1 = new Adder();
     Adder A2 = new Adder();
     Multiplier M = new Multiplier();

     //INTERCONNECT:
     A1.set_a(X1);
     A1.set_b(Y1);
     A2.set_a(X2);
     A2.set_b(Y2);
     M.set_a(A1.get_s());
     M.set_b(A2.get_s());
     P1 = M.get_p();
  }
}
```

Fig. 2. Java description of the objects

2.1. A simple dataflow

Figure 1 shows a simple data flow consisting of two adders and one multiplier. It is clear that these hardware components are running all the time and in parallel. To be able to describe the same behavior in Java, we build a class named *HardwareThread* that inherits from the *Thread*-class. All the Java objects, representing a hardware component, inherit from the *HardwareThread*-class. Thus they are really running in parallel. The class *HardwareThread* contains one method running all the time: *calc()*. This method is empty and can be overwritten by a subclass with a method containing the functionality of the object. It represents the continuous characteristic of the hardware. The *DECLARATION*-part of Figure 2 shows the Java description of the adders and the multiplier.

It is visible, that the inputs and outputs of the functional units in figure 1 are represented by get and set methods as usual in OOP. This emphasizes the similarity between hardware components and objects. Both have well defined input and output ports. In VHDL they are represented by in and out signals of the component, in Java they are represented by set and get methods. The functionality is encapsulated. In VHDL it is part of the body of the component. In Java we implemented it in the method *calc()*. Using threads both are running all the time and parallel to other objects or components. To get the functionality represented in figure 1 the Java objects have to be combined with each other. This is done in the *INTERCONNECT*-part of figure 2. The object named *simpleJava* instantiates the two adders and the multiplier. This is done in the *INSTANTIATION*-part of figure 2.

2.2. Parallel Object Language

The Java description shown in figure 2 can be processed in two ways. First, one can let it run in software, using a usual Java compiler. Second, one can translate it to VHDL, synthesize it and let it run in hardware. Our target was to provide an environment where both executions act identically. Thus one major problem in our first example is the data integrity and serializability. The central question is: how long does it take before all the changes at the outputs caused by the inputs have been done? In hardware this problem is solved using flip-flops and clock signals. The synthesis tool calculates at compile time if the distance between two similar edges of the clock is long enough, so that no glitch is stored in the flip-flops. For the software execution there is no mechanism in figure 2 that ensures that the adders have calculated the new value (based on the inputs X1, X2, Y1 and Y2) before the multiplier uses their output. The same problems exists for the multiplier.

To solve this problem we extended the class *HardwareThread* with methods and signals that ensure the integrity and serializability of the output signals. Using the encapsulation and inheritance of Java objects, the developer does not have to handle these signals. Unfortunately, when the *HardwareThread* class was extended in this way, the elegance of

the descriptions shown in figure 2 was destroyed. There was only one set and one get method each using communication objects, and the inter object communication became much more complex. For instance synchronization keywords had to be implemented. For software execution this just destroys the elegance. For hardware execution it makes the hardware-to-software compilation much more complex, since the compiler has to analyze the Java synchronization keywords and has to translate them correctly to hardware synchronization methods (like flip-flops and clock signals). Due to this we decided to introduce a Java precompiler called POL (Parallel Object Language). POL uses nearly the same syntax as Java, but starting the POL precompiler for software execution results in a Java program enriched with the synchronization signals and methods.

```
public class simplePOL extends ParObj {
   ParObjVec <Adder> Adders = new ParObjVec<Adder>;
   ParObjVec <Multiplier> Multipliers
                        = new ParObjVec<Multiplier>;

   public simplePOL() {
      Adders.add(new Adder());
      Adders.add(new Adder());
      Multipliers.add(new Multiplier());
   }
   class Adder extends ParObj {
      private int a, b, s;
      public set_a(int a) { this.a = a; }
      public set_b(int b) { this.b = b; }
      public get_s() { return this.s; }
      protected void calc() { s = a + b; }
   }
   class Multiplier extends ParObj {
      private int a, b, p;
      public set_a(int a) { this.a = a; }
      public set_b(int b) { this.b = b; }
      public get_p() { return this.p; }
      protected void calc() { p = a * b; }
   }
   public void calc() {
     Adders<0>.set_a(X1);
     Adders<0>.set_b(Y1);
     Adders<1>.set_a(X2);
     Adders<1>.set_b(Y2);
     Multipliers<0>.set_a(Adders<0>.get_s());
     Multipliers<0>.set_b(Adders<1>.get_s());
     P1 = Multipliers<0>.get_p();
  }
}
```

Fig. 3. Parallel Object Language

Figure 3 demonstrates the POL implementation of the simple data flow. POL ensures the data integrity by itself but some design rules have to be considered. Every class that shall be interpreted as Parallel Object has to inherit from *ParObj*. This class can contain set and get methods for the attributes. No public attributes and no direct attribute access is allowed. For inter object communication the get and set methods have to be used. The POL precompiler translates the several set and get methods to one set method and one get method enriched with synchronization elements (as additional classes, methods, attributes and keywords).

Interacting objects are arranged in hierarchical groups. In figure 3 the two adders and the multiplier are packaged in the object *simplePOL*. Every Parallel Object has to be part of a Parallel Object Vector (*ParObjVec*). These vectors contain all objects of the same type. This is crucial to be able to handle dynamically created objects in software and in hardware and will be focused in section 3. The access to the Parallel Objects is handled via the Parallel Object Vectors. The method *calc()* is running permanently and represents the continuous characteristic of hardware components.

Due to the strict restrictions of POL it is possible to translate the POL objects directly to parallel running hardware components. Set methods are translated to input signals. Get methods become output signals and the functionality of the components is extracted from the method *calc()*. Of course POL objects can be more complex than a simple addition or multiplication. The method *calc()* can contain many branches, loops and calculations. Several projects focused the translation of sequential process descriptions into a hardware description language (HDL). These well known methods of process parallelization can be used for the translation of the Java code inside of *calc()*.

3. DYNAMIC OBJECTS

3.1. A simple example: Pong

In the last example the Parallel Objects have been static in the way that a POL-to-hardware compiler can analyze the structure of the POL program shown in figure 3 and recognize that Parallel Objects are only created in the constructor. In this section we want to face the dynamic instantiation of Parallel Objects in POL. For this we use the popular game Pong. Our Pong example consists of 2 user controllable bars and n balls, while n can be dynamically increased and decreased. Bars and balls are both represented by Parallel Objects.

The *DECLARATION*-part of Figure 4 shows the implementation of the Parallel Object *Ball*. Every *Ball* is part of a Parallel Object Vector called *Balls*. The *Bars* are part of a Parallel Object Vector named *Bars*. The command line *"for (Bar b: Bars)"* realizes the access to all *Bars*. This is the POL method to access all objects sharing one Parallel Object Vector. Thus it is possible to generate any number of *Balls* or *Bars* and to communicate with them. For software execution the POL precompiler enriches this access with some synchronization commands.

In figure 4 we introduced the method *finish()*. It is public and can be called by the object itself or by an other object. It changes the behavior of the Parallel Object so that it is not running any longer. Furthermore the object is being removed from the Parallel Object Vector.

The *INSTANTIATION*-part of Figure 4 demonstrates the implementation of the method *calc()* belonging to the object

```
...
//DECLARATION:
class Ball extends ParObj{
   private int x, y, dirx, diry;
   public Ball (int x, int y) {
     this.x = x; this.y = y;
     this.dirx = 1; this.diry = 1;
   }
   public calc()  {
     x = x + dirx;
     y = y + diry;
     if (y<=0) diry = 1;
     if (y>=MAX_Y) diry = -1;

     for (Bar b: Bars)
       if ((x == b.get_x())&&(y == b.get_y()) dirx *= -1;
     if (x<0) finish();
     if (x>MAX_X) finish();
   }
}

//INSTANTIATION:
public void calc() {
     String s;
     s = stdin();
     if (s == "1") Bars<0>.up();
     if (s == "2") Bars<0>.down();
     if (s == "3") Bars<1>.up();
     if (s == "4") Bars<1>.down();
     if (s == "5") Balls.add(new Ball(nX, nY));
}
...
```

Fig. 4. Parallel Objects vs. Pong

that realizes Pong. The command *"Balls.add(new Ball(nX, nY))"* creates a new *Ball* object at position (nX, nY).

3.2. Dynamically Partial Reconfiguration

To be able to translate our Pong game to hardware it is mandatory to be able to instantiate hardware components at runtime. Using normal synthesis tools this is completely impossible, but Xilinx FPGAs like the "Virtex-4" are able to be reconfigured partially and dynamically. Furthermore these FPGAs provide an internal configuration access port (ICAP) that makes it possible to control the partial reconfiguration via the logic on the chip itself. Based on these possibilities the basic idea is to partition the target FPGA into a static and some dynamic areas. The dynamic areas are placeholders for the components that have to be instantiated at runtime (e.g. the components representing the Balls). The static part consists of all components belonging to static objects (e.g. a serial port controller) and the Hardware Scheduler, a program that controls the reconfiguration process. It is the Hardware Scheduler that decides which dynamic component is loaded at which time and to which dynamic area. In a very simple case there are always enough areas to instantiate the dynamic components. In a more complex case there are more dynamic components than dynamic areas and thus the Hardware Scheduler has to instantiate the dynamic components alternately [6].

To be able to describe the complete reconfiguration environment in Java we used the JSB (Java System Builder) [7]. The JSB provides the *ip cores* of the Xilinx EDK as Java objects. Originally it is used to describe static embedded systems on system level (like the EDK does). We extended the JSB, so that it is able to instantiate the Hardware Scheduler and to translate the POL objects to Java or to hardware. For software execution the POL descriptions are translated to ordinary Java files as described before. The static JSB components provide software simulations of their hardware functionality. (For example the class *pushButtons* provides a graphical interface with clickable buttons.) For hardware execution the static JSB components provide their corresponding MSS, MHS and MPD files representing the static part of the design [8]. (For example the class *VGAout* provides a video controller. This is used to visualize the balls and bars on a monitor.) Furthermore the POL description is being translated into static components (like the bars) and dynamic components (like the balls). The static components become a part of the static design, the dynamic components are compiled independently. After generating the MSS, MHS, MPD and VHDL files the Xilinx tools EDK, ISE and PlanAhead are used to generate the static and the dynamic bitfiles. The static bitfile is loaded onto the FPGA. Next the dynamic bitfiles are passed to the Hardware Scheduler running on the FPGA. Now it is possible to create and to remove hardware components dynamically by configuring them into the dynamic areas. Figure 5 demonstrates the software execution of our Pong game, figure 6 shows a screen shot of the hardware execution of our Pong game.

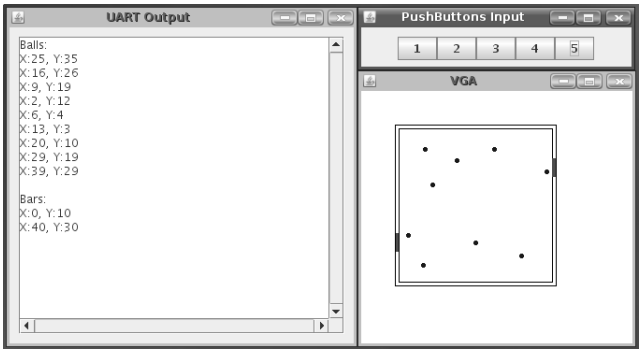

Fig. 5. Software execution of Pong

4. IMPLEMENTATION DETAILS

4.1. Communication

In many applications programmable hardware is used to handle huge data streams. Examples are the data acquisition in detector systems [9], video streaming [10] and networking [11]. Thus the dynamic hardware generated by a POL-to-hardware compiler has to be able to handle such huge data

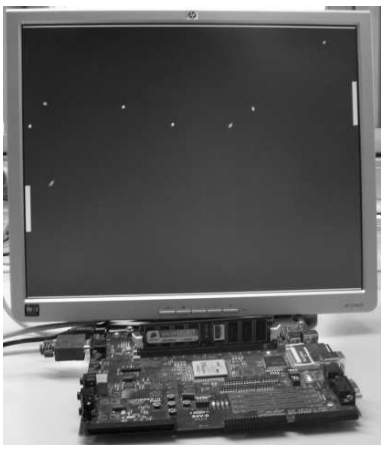

Fig. 6. Hardware execution of Pong

streams. This negates the usage of a simple bus to let the Hardware Objects communicate with each other. Using a bus, the Hardware Objects 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 bottle neck, we developed a communication matrix that provides a parallel inter object communication. The communication matrix is part of the static design and is connected to the dynamic parts via bus macros [12]. It is also connected to the static Hardware Objects. The matrix contains a FIFO pool. Every Hardware Object can use one or more FIFOs of this pool to store its data. The right connection between the Hardware Objects and the FIFOs is established automatically by the POL-to-hardware compiler. For this the compiler generates a unique object number for each Hardware Object. This number is used to address the objects. Before an object is removed it can store its context in a FIFO. In this case, it uses its own object number as the target address. Furthermore, it can store data for other objects in the FIFO pool. For this it uses the number of the target object as the target address. The matrix contains a set of multiplexers that connect the FIFOs to the right Hardware Objects depending on the target addresses. Since the FIFOs can be written and read in parallel, the communication matrix keeps the parallelism. The compiler generates special object numbers representing the Object Vectors (e.g. *Bars* in figure 4). Hardware Objects can use this object number to address all objects belonging to one Object Vector.

Since in POL objects can dynamically instantiate other objects, Hardware Objects have to be able to instantiate other Hardware Objects. For this the bus macros contain an instantiation bus. Using this bus, a Hardware Object can tell the Scheduler the object number of the new Hardware Object that shall be instantiated.

4.2. Limitations

The flexibility of a dynamically reconfigurable System using Hardware Objects is limited by the used hardware. A very important threshold is the number of FIFOs the communication matrix contains. Depending on the functionality of an object the compiler has to provide one, two or even more FIFOs to one Hardware Object to maintain the parallelism. For this reason, the maximum parallelism is determined by the number of parallel accessible FIFOs. In an environment with n object areas one has to implement at least $n + 2$ FIFOs. One output FIFO for every Hardware Object and two FIFOs as connection between the communication matrix and the outer world.

If there are more active Hardware Objects than object areas, the Scheduler can load the Hardware Objects alternatively. When Hardware Object A sends data to the physically removed Hardware Object B this data needs to be stored until B is loaded again. Thus the size of the FIFOs divided through the data rate of A determines the time B can be removed until A is not able to store anymore data and is blocked. Thus the maximum number of Hardware Objects running at the same time is limited by the number of object areas and the needed data rates. The higher the data rates are, the less reconfigurations can be performed.

5. APPLICATIONS

To evaluate the possibilities and the limitations of a POL-to-hardware compiler we implemented three examples. The first one was Pong (implemented on a Virtex-2 Pro 30), demonstrating the possibility to add and to remove Hardware Objects dynamically. It has already been focused in section 3. The other two example applications are an audio DSP and video processing (implemented on a Virtex-5 LX50), demonstrating the behavior of a dynamic environment regarding data streams.

Our audio DSP has a 32 bit input (16 bit for the left and 16 bit for the right channel) and a 32 bit output. We implemented 4 objects representing 4 different effects: a high pass, a low pass, a distortion and an echo. Every object was instantiated to activate the corresponding effect and removed to deactivate it. To test the limits of the communication matrix we only provided one dynamic area. Therefore, if the user activates all 4 objects, the Scheduler has to configure all the 4 objects in turn. Nevertheless, our target was an uninterrupted audio stream at the output. This was possible since the Hardware Objects could calculate 100.000 samples in one millisecond (due to the clock of 100MHz), but only 48 samples per millisecond were needed (due to the audio quality). The Scheduler needed 0.2 ms to reconfigure the dynamic area. Knowing this parameter the compiler calculated that one Hardware Object only had to stay configured for 1 microsecond and thus the FIFO has to store at

least 100 samples. Thus, every Hardware Object calculated 1 microsecond and paused 803 microseconds. The compiler instantiated 4 FIFOs for the 4 objects, each with a size of 512 Byte. If all 4 effects were activated, the resulting system loaded the 4 Hardware Objects alternately using DPR, but due to the FIFOs the audio stream stayed uninterrupted the whole time. The latency was 2.8 ms.

In our audio example one reconfiguration turn lasted 804 microseconds, but only 4 microseconds of a turn were used for calculation. 800 microseconds were used for reconfiguration. The reason is the very low speed of the Xilinx reconfiguration interface. The 4 microseconds calculation time sufficed, since only 48 samples per millisecond were needed, but 100.000 samples could be processed in one millisecond. If the number of needed dates per second rises, the calculation time and with it the needed FIFO size rises, too. To demonstrate this, we will take a look at our third example: video processing. We implemented a Hardware Object realizing an edge detection and a Hardware Object realizing a gamma correction in a video stream. Our stream consisted of 25 frames per second. Every frame had a width of 352 and a height of 288 Pixels. One Pixel had the size of 3 bytes. Thus, we needed a data rate of 7.6 MB/s. The filters could produce data with a data rate of 25 MB/s. To be able to load the two filters in turns without interrupting the stream, every object had to calculate about 0.3 ms. In this time it produces 2.3 MB data. We therefore needed a minimum of 4 FIFOs with a size of 2.3 MB. This exceeded the limitations of our FPGA. Hence it was not possible to configure the two Hardware Objects alternately without interrupting the video stream. This points out that a permanent reconfiguration as realized in our audio example is not always possible. Nevertheless, in environments with huge data streams (like video processing or data acquisition) the reconfiguration can be used to exchange parts of the processing system to be able to react to changing requirements [10]. In this case DPR can help to save chip resources, since not every Hardware Object has to be instantiated from the beginning. Objects are only instantiated when they are needed. The number of needed dynamic areas is determined by the maximum number of Hardware Objects running in parallel and producing huge data streams.

6. CONCLUSION

In this paper we presented POL as one possibility to do a parallel object description. Our POL-to-hardware compiler translates POL objects directly to Hardware Objects. The *get* and the *set* methods become the inputs and the outputs and the content of the method *calc()* determines the functionality of the resulting Hardware Object. Since the target hardware is partial and dynamically reconfigurable, the dynamic instantiation of objects does not have to be avoided, but can be directly implemented. This combination of object-orientation and DPR solves two problems at once. First, the dynamic character of OOP does not have to be removed, but can be translated directly to the hardware. Therefore, the generated Hardware Objects are very similar to their software pendants. Hence, it is no longer necessary to translate object-oriented programs to a sequential process description (like Java bytecode) and then to reparallelize this sequential description. Second, POL provides a very elegant way to control dynamic hardware. Today it is still very complex to use DPR, since the developer has to understand the reconfiguration techniques in detail to be able to use DPR. Using POL the complete DPR techniques (like the instantiation of bus macros, the partitioning of the chip, the instantiation of a scheduler, the instantiation of a communication matrix including FIFOs) are encapsulated. The instantiation of a new Hardware Object is done with a simple *new*. Due to these new possibilities, which come with the combination of DPR and OOP, we are convinced that it is time to reconsider the use of OOP for hardware design.

7. REFERENCES

[1] D. Morris, D. Evansa, and P. Green, "Object oriented computer system engineering," *Springer-Verlag*, 1996.

[2] A. C. S. Becka and G. Gaydadjiev, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proc. DATE*, 2008, pp. 1208–1213.

[3] NVIDIA-Corporation, "Nvidia cuda computed device architecture programming guide version 1.1," Nov. 2007.

[4] M. Edwards and P. Green, "An object oriented design method for reconfigurable computing systems," in *Proc. DATE*, 2000.

[5] B. Hutchings and M. Rytting, "A cad suite for high-performance fpga design," in *Field-Programmable Custom Computing Machines*.

[6] N. Abel, "Schnelle dynamische partielle rekonfiguration in hardware mit inter-task-kommunikation," *University of Leipzig*, June 2005.

[7] J. Gebelein, "System-specification of embedded systems in java for synthesis," *University of Leipzig*, July 2007.

[8] Xilinx, "Platform specification format reference manual," *www.xilinx.com*, 2007.

[9] T. Alt and V. Lindenstruth, "Fpga based pre-/coprocessors for the alice hlt," in *Proc. DPG-Conference*, Mar. 2005.

[10] C. Claus and J. Zeppenfeld, "Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system," in *Proc. DATE*, 2007.

[11] S. Saponara and E. Petri, "Fpga-based networking systems for high data-rate and reliable in-vehicle communications," in *Proc. DATE*, 2007.

[12] P. Lysaght and B. Blodget, "Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," in *Proc. FPL*, 2006, pp. 012–017.