PARALLEL HARDWARE OBJECTS FOR DYNAMICALLY PARTIAL RECONFIGURATION

Norbert Abel, Frederik Grüll, Nick Meier, Andreas Beyer, 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 dataflow 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). Software 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 themselves 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 instantiable Hardware Objects.

2. HARDWARE OBJECTS

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

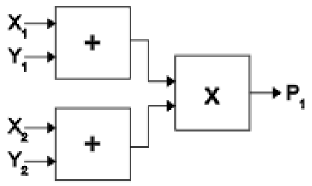

Fig. 1. A simple dataflow

2.1. A simple dataflow in Java

Figure 1 shows a simple dataflow 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. Signal Integrity

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

```
//DECLARATION:
class Adder extends HardwareThread {
   private int a, b, s;
   public void set_a(int a) { this.a = a; }
   public void set_b(int b) { this.b = b; }
   public int get_s() { return this.s; }
   protected void calc() { s = a + b; }
}
class Multiplier extends HardwareThread {
   private int a, b, p;
   public void set_a(int a) { this.a = a; }
   public void set_b(int b) { this.b = b; }
   public int 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

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.

2.3. The Reconfiguration Framework

Before we take a closer look on POL, we want to look at the functionality of our Reconfiguration Framework, since the way POL is specified is the result of the way the target hardware is designed. To be able to translate Java objects directly 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. The static part consists of all components belonging to static objects 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].

In many applications programmable hardware is used to handle huge data streams. Examples are the data acquisition in detector systems [7] and video streaming [8]. 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 Objects communicate with each other. Using a bus, the Hardware Objects would be calculating in parallel but would have to provide the results of their calculations 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 [9]. It is also connected to the static Hardware Objects. The matrix contains a FIFO pool, consisting of one FIFO per Hardware Object Class and two System FIFOs (in and out). Furthermore the matrix contains a set of multiplexers, connecting the FIFOs with the related dynamic areas. Due to the FIFOs the communication is pipelined. This allows a higher parallelism (even if the objects are loaded alternately), but it also determines the way the tasks are communicating with each other. *Set* operations put one entry into a FIFO. *Get* operations take one entry out of a FIFO. POL has to be designed in a way that complies with this architecture.

3. POL

Figure 3 demonstrates the POL implementation of the simple dataflow. 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*. The communication between the objects is realized with *signals* and *slots* (inspired by Qt [10]). No public attributes and no direct attribute access is 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 components. It is the only method containing functionality.

```
import "cosinus.vhd";

class simplePOL extends ParObj {
  Multiplier M; Adder A1; Adder A2; Cosinus C;

  class Multiplier extends ParObj {
    Slot a, b; Signal p;
    public void calc(){p.emit(a.get()*b.get());}
  }
  class Adder extends ParObj {
    Slot a, b; Signal s;
    public void calc(){s.emit(a.get()+b.get());}
  }
  class Cosinus extends ParObj {
    Slot a; Signal c; int result;
    public void calc(){
      component cosinus (in=>a.get(), out=>result);
      c.emit(result);
    }
  }

  simplePOL(){
    M = new Multiplier();
    A1 = new Adder(); A2 = new Adder();
    M.p.connect(Out[0]);
    A1.s.connect(M.a);
    A2.s.connect(M.b);
    In[0].connect(A1.a);
    In[1].connect(A1.b);
    In[2].connect(A2.a);
    In[3].connect(A2.b);
  }

  public void calc() {
    int s = In[4].get(0);
    if (s == 1) {
      C = new Cosinus();
      M.p.disconnect(Out[0]);
      M.p.connect(C.a);
      C.c.connect(Out[0]);
    } else if (s == 2) {
      M.p.disconnect(C.a);
      C.c.disconnect(Out[0]);
      M.p.connect(Out[0]);
      C.finish();
    }
  }
}
```

Fig. 3. Parallel Object Language

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 an VHDL input signal. Every POL *signal* becomes an VHDL output signal and the functionality of the components is extracted from the method *calc()*. The method *connect* tells the communication matrix, which VHDL output signal is connected to which VHDL input signal via the FIFOs. In contrast to the solution in Figure 2 the objects are directly communicating to each other without the usage of an additional "communication object". This is mandatory to enable a multi parallel inter object communication. The method *emit* puts one entry into a FIFO, *get* takes one entry out of the FIFO. There are two versions of *get*. *Get()* is blocking in case of an empty FIFO. *Get(default)* is not blocking and returns *default* in case of an empty FIFO.

In figure 3 the Parallel Objects A1, A2 and M are static in the way that a POL-to-hardware compiler can analyze the structure of the POL program and recognize that these Parallel Objects are only created in the constructor and not destroyed at runtime. Hence, we added the class *Cosinus* to demonstrate the dynamic instantiation of Parallel Objects in POL. If the fifth system input slot contains a 1, an instance of this class is instantiated at runtime. To describe the instantiation of the *Cosinus* object in POL it is just necessary to use the command *new*. The POL-to-hardware compiler translates this *new* to an dynamic bus command. This bus connects the hardware objects with the scheduler. It can be used to create and to destroy object instances. M1 and the system output are disconnected using the method *disconnect*. Afterwards the Input of C1 is connected to the output of M1 and the output of C1 is connected to the system output. Thus, the dataflow shown in figure 1 has been extended by a cosinus calculation after the multiplication.

Of course POL objects can be more complex than a simple addition or multiplication. The code in figure 3 just demonstrates the functionality of POL. Possible real-world applications are filters (like for video streaming or data acquisition) that have to be exchanged on demand. For this the method *calc()* can contain many branches, loops and calculations. The POL-to-hardware compiler is able to translate this Java code to VHDL. Furthermore, for better re-use of existing code, it is possible to include VHDL components into POL. This is done using the key word *component*. It looks and works like the *component* key word in VHDL. All VHDL components used in POL have to provide an std_logic input called *Strobe_In*, an std_logic output called *Strobe_Out* and a clock input called *CLK*. *Strobe_In* tells the VHDL component to start its calculation. *Strobe_Out* is used by the VHDL component to confirm its outputs.

In order to remove objects 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. Afterwards it is removed from the scheduling list. If the fifth system input slot in figure 3 contains a 2, the cosinus is removed from the dataflow.

4. 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 *slots* and the *signals* 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.

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

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

[9] 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.

[10] Trolltech, "Online reference documentation," *doc.trolltech.com*, 2008.