Abstract

Dynamic Partial Reconfiguration (DPR) is a promising technology ready for use, enabling the design of more flexible and efficient systems. However, existing design flows for DPR are either low-level and complex or lack support for automatic synthesis. In this paper, we present a SystemC based modelling and synthesis flow using the OSSS+R framework for reconfigurable systems. Our approach addresses reconfiguration already on application level enabling early exploration and analysis of the effects of DPR. Moreover it also allows quick implementation of such systems using our automatic synthesis flow. We demonstrate our approach using an educational example.

1. Introduction

Dynamic Partial Reconfiguration (DPR) is the ability of FPGAs, to change some parts of their programming while the remaining (static) parts keep operating. With current hardware description languages like Verilog, VHDL, or SystemC [7], DPR can only be expressed at a very low implementation level. There is no support for explicitly expressing the change of design components at runtime in these HDLs. More importantly, it is not possible to explore the impact of reconfigurable sub-systems on the performance and behaviour of the system as a whole in early design phases.

Additionally, using DPR manually requires a lot additional design effort, since it affects both static and dynamic parts of the design. This time-consuming and error-prone work makes DPR prohibitive for practical use in real products. There has been quite some research to overcome this. Unfortunately the proposed solutions required a significant change in modelling the static part of the design.

OSSS+R is a SystemC based design methodology enabling algorithmic specification in C/C++, functional simulation and automated synthesis. Our extension to the set of available modelling primitives and simulation abilities is done in terms of a SystemC domain-specific library, available under the LGPL license. Simulation can be done with any IEEE 1666-2005 standard conforming simulator. The designer identifies potential candidates for dynamic reconfiguration, marks them and observes the effects by simulation. The model can be directly fed into the Fossy synthesis tool, generating VHDL. Feeding resulting files into an FPGA synthesis tool quickly yields bitfiles and initial, approximate configuration times. A back-annotation of these times into the abstract model allows performance evaluations.

The basics of our approach have been presented in [15, 16] leaving the synthesis as an outlook. This contribution fills the gap by describing the complete design flow including automatic synthesis. We demonstrate our approach using a reconfigurable waveform generator, starting from a pure C++ application level model which is refined to a high level OSSS+R model and finally synthesised to a register transfer level model. We further present results of the final implementation of the system on a Xilinx ML-401 development board using the Xilinx Early Access Partial Reconfiguration design flow [18].

2. Previous work

OSSS [5] is a SystemC-based design methodology enabling the object-oriented modelling of synthesisable hardware/software systems. The synthesisable subset of SystemC is extended by additional elements for high-level modelling, like shared variables, polymorphism or transaction level modelling. It consists of a simulation library and a synthesis tool, called Fossy [6].

OSSS+R extends OSSS by adding language elements for reconfigurable components and component arbitration. The
approach uses object-oriented techniques as an abstraction mechanism for dynamic partial reconfiguration. Reconfigurable components are modelled as polymorphic objects, providing a statically determined method-interface to the system. The polymorphism allows to exchange the currently available behaviour during run-time. Additionally, activity within reconfigurable components is limited to the execution of its methods. Therefore, the disabling of a component is well-defined outside those periods of activity.

Given this abstract modelling most of the low-level details of the reconfiguration are transparent to the designer. On the other hand, the simulation library is able to reflect the effects of reconfiguration such as reconfiguration delays and concurrent accesses to a single reconfiguration controller. Therefore, designers can explore whether or not reconfiguration is beneficial for their application already in early phases of the design cycle.

A drawback is, that not all general reconfigurable circuitry can be modeled. For example, user-defined processes can’t be embedded in reconfigurable parts. Instead, reconfigurable areas are shared and flexible datapath extensions that are automatically managed.

3. Related work

There are other frameworks which allow both modelling and syntheising of dynamic reconfigurable systems.

One example is Pebble[10], a low level HDL providing specific statements for reconfiguration. One is a mux/demux encapsulation of logic variants. The control inputs of these muxes are used as reconfiguration conditions. The logic variants are to be exchanged during reconfiguration. Additionally, a RECONFIGURE_IF statement allows an alternative specification. The specification does not cover reconfiguration times. A compiled model can be simulated using the Rebecca simulator. The authors demonstrated synthesis for an Xilinx 6200 FPGA.

JHDL[3] is a structural hardware description language based on Java. It provides reconfigurable elements, called PRSocket, which can receive a Reconfigure(int) call, requesting a specified implementation. Depending on the argument, new circuit nodes are created. JHDL can be simulated and synthesised. The system clock needs to be stopped during reconfiguration, which makes modelling of reconfiguration times impossible.

T.K. Lee et al. [9] used RT C to describe a reconfigurable system. The reconfigurable elements are tasks, which are grouped in structs, arrays or unions. The grouping determines the replacement, e.g. members of a union are mutually exclusive. These groupings allow influence on control complexity, area demands and design performance. An example model was transformed into Handel-C and RTPebble, with tool assistance and some manual work. It was then implemented on a Celoxica RC1000-PP board.

The DCS toolset [11] accepts specially crafted VHDL as its input, containing all implementations of the configurable components. It also needs auxiliary scheduling and timing information. The toolset then generates a simulation model for debugging. Since the design is already given in VHDL, FPGA vendor tools are used to implement the design [14].

There are other approaches, based on SystemC (like OSSS+R), which allow simulation but do not have no tool-assisted synthesis.

SyCERS[1] is a framework allowing modelling of run-time reconfiguration, intended to explore design alternatives. The functionality to be replaced is represented by functions inside modules. These functions are called from the body of SC_THREADS and SC_METHODs. By using function pointers to change the function at simulation time the dynamic behaviour is achieved. Simulation is done by mapping to the Caronte architecture, containing a microprocessor to access the reconfigurable modules.

In [2] a modelling framework using dynamic thread spawning is used. The framework implements an additional layer to the SystemC kernel providing required features like dynamic ports. Using this layer, threads can be replaced at runtime, expressing dynamic behaviour.

The ReChannel library[13] allows modelling of run-time reconfiguration at different levels of abstraction. Though ReChannel guides the designer during iterative refinement to lower levels of abstraction it does not provide automatic synthesis.

In these approaches, the management of the dynamic resources (which thread may use which resources at what instant) needs to be specified manually. An exception to this is used in the ADRIATIC project [12]. In this approach, mutual exclusive modules are to be described as bus slaves. Then multiple slaves are grouped and wrapped in a dynamic reconfigurable fabric (DRCF) which switches among them and acts as a physical bus slave itself. The bus master requests a specific logical slave by its bus address which is then utilised by the DRCF to enable the requested logical bus slave. The DRCF introduction is done at RT level.

The given list of SystemC-based approaches do not include those requiring a modified simulation kernel. Further SystemC-based approaches can be found in [4, 8, 17].

4. Modelling example: A waveform generator

The guiding example for this paper is a C++ benchmark implementing an audio signal generator. It contains a generator function producing three different kinds of waveforms: triangle, square and sawtooth. The waveforms are amplified by the output of an envelope generator and then processed by either a simple low pass filter or an periodic amplifier.
which increases and decreases the volume over time. The waveform type and filter kind may be reconfigured.

The original model has been implemented as an object-oriented C++ model. Each type of the waveform generator module is implemented as individual class with one common base class, since they are all waveform generators. Same applies to the selectable filters. The amplitude of the envelope generator object can be set via a method call.

This example is rather simple and artificial. However, the design is easy to understand and well-suited to explain problems and solutions of DPR and to illustrate the overall OSSS+R design flow.

5. Modelling in OSSS+R

In a SystemC design flow the design entry may be a C/C++ description of the application’s core algorithms. This is to be refined into a hardware description, using SystemC modelling elements. SystemC presents itself as a library, not a language, so one may also use all features of C/C++. While this allows faster simulation and easier modelling, the drawback is, that such a model might use features which are (typically) not synthesisable. Manual recoding would be required to obtain a synthesisable model written in Verilog or VHDL for example. A designer may be tempted to avoid all non-synthesisable features in the first place, however this would sacrifice advantages like more abstract modelling.

OSSS+R encourages the designer to use C++ features, since classes, objects and inheritance are synthesisable. For more complex components, like concurrently used objects, OSSS+R provides synthesisable containers to reduce the dilemma described before. A group of objects where each member is accessed rarely overlapped with other members of the same group is a good candidate for reconfiguration. Additionally, polymorphic pointers in the C++ model are a hint to dynamic objects which also make good candidates.

In the initial C++ model of the waveform generator some objects were implemented using polymorphism. The different implementations of the abstract generator interface were accessed through a polymorphic pointer. This way, the referenced generator could easily be switched from one waveform generator object to another without having to change the interface to the object. These generator objects are used mutually exclusive and switches are rare events. To let the generators share the same physical reconfigurable area the polymorphic pointer is replaced by an OSSS+R reconfigurable object. The reconfigurable object uses the same generator base class as the polymorphic pointer and provides the same C++ syntax for accessing the object. For the same reason as the waveform generators are identified, the filters are good candidates for reconfiguration, too. They are mapped to a second reconfigurable object, resulting in two reconfigurable areas within the final system. The white boxes in Figure 1 show the application after its transformation to OSSS+R.

**Code Transformations.** The C++ model contains a polymorphic pointer `wg` which is initialised using one of the available generator classes.

```cpp
WaveformGenerator * wg;
wg = new SquareGenerator();
sample next_sample = wg->fetch();
```

For OSSS+R, this pointer is moved inside the SystemC module `WaveGenModule` and transformed into a reconfigurable object:

```cpp
SC_MODULE( WaveGenModule ) {
  // ... SystemC code here
  ossss_recon< WaveformGenerator > wg;
}
```

Within the module constructor, the reconfigurable object is bound to the process `work`, enabling `work` to access the
object. Similarly, the object could be bound to even more processes. To resolve concurrent accesses by more than one process, the reconfigurable object automatically provides a built-in scheduler, serialising all incoming requests.

As it can be seen in the implementation of the process work, the new variable is used almost like the original C++ pointer, the only difference being the assignment to an object, where new is omitted.

```c
void WaveGenModule::work() {
    // ... SystemC code here ...
    while (true) {
        switch (generator.read()) {
            case 0: wg = SquareGenerator(); break;
            case 1: wg = SawtoothGenerator(); break;
            case 2: ...
        }
        sample next_sample = wg->fetch();
        // ... SystemC code here ...
    }
}
```

The resulting implementation performs a reconfiguration, whenever the runtime class of wg changes, possibly caused by an assignment. However, if the run-time class matches the previous one, only the object’s attributes are modified. State preservation could have been obtained by back-annotating to the initial model, providing the exact timing of the partial bitstreams and the performance of the chosen reconfiguration controller. These timings can then be back-annotated into the OSSS+R model.

### Devices and Timing

To reflect reconfiguration and context switch times during simulation with proper timing, OSSS+R supports timing annotations provided by the designer. Timing annotations are defined as part of the target platform definition. They are given for a combination of platform and class type, e.g. Virtex 4 and Sawtooth. Designers may specify the time needed for a reconfiguration and the time needed to store the state of a class instance.

```c
OSSS_DECLARE_TIME(    // Timing:
    virtex4,         // Platform
    Sawtooth,        // Class name
    sc_time( 2, SC_US),   // Context save/restore
    sc_time(100, SC_US));  // Reconfiguration time
```

Initially during the modelling phase, the specified times are rough estimates by the designer. Later on, when the final implementations of the configurations are available, the exact reconfiguration times can be obtained through the size of the partial bitstreams and the performance of the chosen reconfiguration controller. These timings can then be back-annotated to the initial model, providing the exact timing behaviour within the application model. If the model shows some unexpected or unwanted behaviour due to this reconfiguration times, these issues can be traced back to the OSSS+R model. This is much more convenient than debugging RT level code.

6. Synthesising OSSS+R

Figure 2 presents the flow from an OSSS+R model to a final FPGA implementation. Initially, the OSSS+R model is simulated to validate its behaviour using the OSSS+R simulation library. The model is then automatically synthesised to register transfer level (RTL) using the Fossy tool. First, OSSS+R specific language elements are replaced with equivalents composed of SystemC components. Then synthesis of the resulting SystemC model to RTL is performed, including class tree synthesis, implicit to explicit FSM transformation etc. The output can be either SystemC or VHDL. The generated VHDL may then be further processed by FPGA vendor tools, e.g. the Xilinx ISE tool suite. Once bitstreams are obtained, the reconfiguration times can be calculated and backannotated into the OSSS+R design.

In Figure 1 a block-diagram of the generated RTL architecture of the waveform generator is shown. The square, gray boxes are generated by Fossy representing infrastructure components which are needed to implement the dynamic partial reconfiguration. While these components are automatically provided and instantiated as simulation models by the simulation library, they are not synthesisable as such and have to be replaced by synthesisable equivalents.

**Recon-Object.** For each reconfigurable object a corresponding reconfigurable area, called slot is generated. This
slot may take any of the classes that have been mapped to
the reconfigurable object. Each of the classes is generated
as a stand-alone module, communicating with other mod-
ules by a signal-based protocol. Thanks to their polymor-
phic nature, all of the classes within one reconfigurable ob-
ject can share the same physical interface although the sig-
nal interpretation varies during runtime. After implement-
ation, each of the classes will be represented by its own
partial bitstream.

Each slot is managed by an accompanying component
controlling the access to the slot, detecting needs for recon-
figurations and initiating reconfiguration requests. Opera-
tions on the reconfigurable object, e.g. method calls on a
signal inside the waveform generator module, are replaced by
a signal-based protocol to the access controller and the slot.
Each request to a reconfigurable object is first directed to
the access controller, to schedule it with other pending re-
quests. If the access is granted and the requested configu-
ration is activated the process directly communicates with
the slot.

Reconfiguration controller. If an access controller de-
tects the need to perform a reconfiguration, a request for
reconfiguration is sent to the platform independent part of
the reconfiguration controller (PIRC). The PIRC is auto-
matically generated by Fossy. In our example, there are
two access controllers requesting services, so the PIRC is
equipped with a scheduler to resolve conflicts. Addition-
ally, the PIRC translates requested class types and location
information to bitstream numbers. The translated requests
are serviced by the platform dependent reconfiguration
controller (PDRC) part. A PDRC is implemented manually
once for a given platform, e.g. a FPGA prototyping board,
and can be re-used for multiple applications. Platform de-
pendent blocks in Figure 1 are shown in black.

Method calls. The user processes contain accesses
(method calls and assignments) to reconfigurable objects
and their contexts. These accesses are replaced by a sig-
nal level protocol between user processes and access con-
trollers (for permission handling and reconfiguration) and
user processes and slots (for method calls and assignments).
In a reconfigurable system, a single user process may com-
municate with a set of different slot implementations, each
having their individual interface signal interpretation. Due
to the strong type system in the original model, it is guar-
anteed that the user process always uses the correct signal
interpretation. After the replacement of all OSSS+R spe-
cific elements with synthesisable SystemC equivalents, the
RTL model is generated as SystemC or VHDL.

RTL simulation model. Typically, a designer wants to
check the result of any automatic transformation by simu-
lating its result. However as standard HDLs do not support
the expression of DPR, the result of the OSSS+R synthesis
cannot be simulated as such. As a solution to this, Fossy
can generate an RTL simulation model, which can be sim-
ulated with any standard HDL simulator. In this model, all
possible configurations of a slot are instantiated in parallel
and connected to a multiplexer structure. For the simula-
tion model, a pseudo PDRC is generated which controls the
select inputs of the multiplexers. For each reconfigura-
tion request, instead of writing bitstreams to an FPGA config-
uration port, the PDRC mimics the behaviour by waiting
for as long as the configuration would take in the real sys-
tem. The waiting time is taken from the timing specifica-
tions which have been given by the designer in the original
model (see Section 5). After a first implementation these
values may also be replaced with the reconfiguration times
of the final partial bitstreams. The pseudo PDRC provides
the same interface as the original, so despite the multiplexer
structure, the rest of the model is identical to the synthesis
model. This way the application can be simulated with stan-
dard HDL simulators and will show the same behaviour as
the reconfigurable design.

From RTL to bitstreams. If the simulation is success-
fully validated using the RT level simulation model, the
RT level synthesis model can be transformed to gate level
and bitstreams using FPGA vendor tools. The synthesis of
OSSS+R has been developed to be platform independent.
However, to support the DPR features of a target platform,
the model usually has to be tailored to a vendor specific
tool framework. Typically, this includes creating a specific
top level, some pinout description files, a floorplanning file
etc. We have implemented this vendor specific adaption for
the Early Access Partial Reconfiguration Flow (EAPR) [18]
from Xilinx.

7. Evaluation

Using the EAPR flow, we have successfully imple-
mented the generated RTL model of the waveform gener-
ator on an ML401 development board from Xilinx. The
PDRC has been designed manually, using the Virtex4 ICAP
directly with a maximum bandwidth of roughly 600 MBit-
s/sec. Table 1 shows the size of the partial bitstreams and
their resulting reconfiguration times.

To get a picture of the overhead introduced by the recon-
figuration infrastructure Table 2 shows the usage of FPGA
resources for PIRC, PDRC and access controllers. Com-
pared to the total resources of the FPGA the overhead is
rather small. While the overhead for PDRC and PIRC is
constant, the resource usage for access controllers would
increase with the number of slots and the use of a sched-
uler. However, we consider the overhead acceptable given
Table 1. Size and configuration time of partial bitstreams

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>configuration time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Amplify</td>
<td>73 155 Bytes</td>
<td>91.44 us</td>
</tr>
<tr>
<td>LowPass</td>
<td>75 414 Bytes</td>
<td>94.27 us</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>73 777 Bytes</td>
<td>92.22 us</td>
</tr>
<tr>
<td>Square</td>
<td>77 195 Bytes</td>
<td>96.49 us</td>
</tr>
<tr>
<td>Triangle</td>
<td>75 093 Bytes</td>
<td>93.87 us</td>
</tr>
</tbody>
</table>

Table 2. Resource usage of infrastructure

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>device utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDRC</td>
<td>209</td>
<td>1 %</td>
</tr>
<tr>
<td>PIRC</td>
<td>289</td>
<td>1.3 %</td>
</tr>
<tr>
<td>Access CTRL Filter</td>
<td>29</td>
<td>0.1 %</td>
</tr>
<tr>
<td>Access CTRL Generator</td>
<td>31</td>
<td>0.1 %</td>
</tr>
</tbody>
</table>

the potential save of FPGA area through the use of DPR.

8. Conclusion and Future Work

In this paper we presented a complete modelling and synthesis flow for DPR systems based on the modelling framework OSSS+R. Using an educational example we demonstrated how a designer can efficiently design such systems without having to deal with the implementation details of DPR. Using the abstraction mechanism of polymorphism, reconfiguration can easily be expressed and captured already on application level. OSSS+R models are synthesised to RTL models using the synthesis tool Fossy. Using the Xilinx EAPR flow, we were able show that the overhead introduced by the DPR infrastructure is acceptable.

In future we will extend Fossy to support the synthesis of reconfigurable objects with Named Contexts [16]. We are also planning to integrate a more flexible approach for the implementation of the communication infrastructure between processes and reconfigurable objects.

References


