System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC

Antti Pelkonen
VTT Electronics
P.O .Box 1100, FIN-90571
Oulu, Finland
antti.pelkonen@vtt.fi

Kostas Masselos
INTRACOM SA
P.O. Box 68, GR-19002
Peania, Attika, Greece
kmas@intracom.gr

Miroslav Cupák
IMEC
Kapeldreef 75, B 3001
Leuven, Belgium
cupac@imec.be

Abstract

To cope with the increasing demand for higher computational power and flexibility, dynamically re-configurable blocks become an important part inside a system-on-chip. Several methods have been proposed to incorporate their reconfiguration aspects in to a design flow. They all lack either an interface to commercially available and industrially used tools or are restricted to a single vendor or technology environment. Therefore a methodology for modeling of dynamically re-configurable blocks at the system-level using SystemC 2.0 is presented. The high-level model is based on a multi-context representation of the different functionalities that will be mapped on the re-configurable block during different run-time periods. By specifying the estimated times of context-switching and active-running in the selected functionality modes, the methodology allows to do true design space exploration at the system-level, without the need to map the design first to an actual technology implementation.

1. Introduction

With the rising performance available for System-on-Chip (SoC) designers and simultaneously rising manufacturing costs, there is a pressing financial reasons for adding more flexibility to the SoC designs without sacrificing the performance and possible parallelism of ASIC. Reconfigurable technologies [1, 2] are viewed as the solution for getting this flexibility, but the introduction of dynamic reconfiguration (also known as run-time reconfiguration) adds new dimensions to the design space of a SoC designer, since the same area can be configured for different functions at different times.

The cost-efficient design of dynamically reconfigurable SoCs require support from tools and methodologies for all abstraction layers and design phases. So far, there has been a absence of industrially adaptable methodologies and tools for system-level design and especially design space exploration. Also the problem with the methodologies presented by the academia is that they can not be adapted to the used design flows of the industry without substantial modifications to the re-usable code base of a company. They are also often bound to single implementation technology.

A large portion of the SoC designs using dynamically reconfigurable hardware are not designed from scratch. The starting point of such designs are often a previous version of the SoC with some more features (Figure 1a). The goal of the design is to implement all the old features and some new ones and implement some parts of the application on dynamically reconfigurable hardware (Figure 1b).

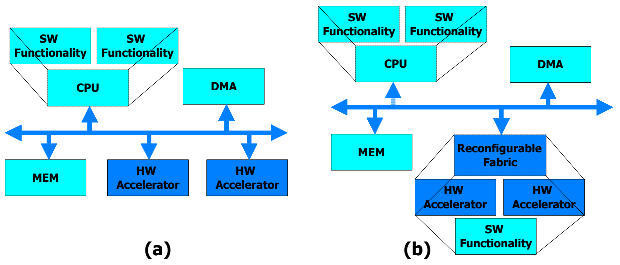

Figure 1. (a) Typical SoC architecture and (b) the modified architecture using dynamically reconfigurable hardware.

This paper presents a system-level modeling methodology and associated tools capable of doing quick design space exploration and which takes in to account the special properties of dynamically reconfigurable hardware. The methodology is based on SystemC, which is rapidly becoming the language of choice for system-level design. This is partly due to the fact that all large EDA vendors support or plan to support SystemC in their tools.

The rest of the paper is organized as follows: In Chapter 2, dynamic reconfiguration is defined and the benefits and drawbacks of its use are identified. In Chapter 3, classes of reconfigurable technologies are presented with case examples. Chapter 4 presents a quick overview of related research and moderns industrially used system-level co-design tools. In Chapter 5, the modeling methodology is presented along with description of a co-design flow developed in projects ADRIATIC. In Sub-Chapter 5.4 and in chapter 6, some limitations and the need of further work is identified.

2. Dynamic reconfigurability

Reconfigurable hardware aims at bridging the gap between implementation efficiency and flexibility as shown in Figure 2 [3]. This is achieved since reconfigurable hardware combines the capability for post-fabrication functionality modification (not present in conventional ASICs) with the spatial/parallel computation style (not present in instruction set processors).

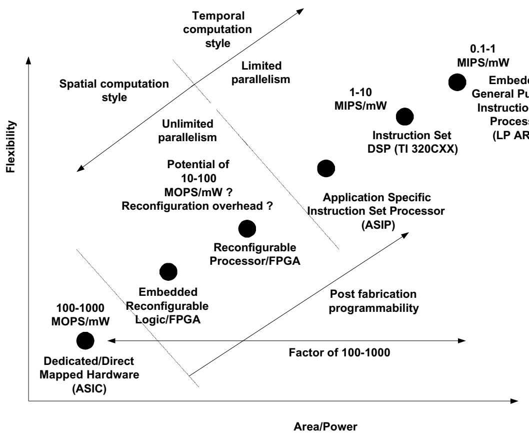
Temporal computation style
Spatial computation style
Limited parallelism
Unlimited parallelism
0.1-1 MIPS/mW
Embedded General Purpose Instruction Set Processor (LP ARM)
1-10 MIPS/mW
Instruction Set DSP (TI 320CXX)
Potential of 10-100 MOPS/mW ? Reconfiguration overhead ?
Application Specific Instruction Set Processor (ASIP)
Reconfigurable Processor/FPGA
Embedded Reconfigurable Logic/FPGA
100-1000 MOPS/mW
Post fabrication programmability
Factor of 100-1000
Dedicated/Direct Mapped Hardware (ASIC)
Flexibility
Area/Power

Figure 2 Flexibility versus implementation efficiency for different architectural styles.

Reconfigurable architectures can be classified with respect to the following parameters:

a) reconfiguration scheme,

b) coupling to a host microprocessor, and

c) granularity of processing elements.

Reconfiguration scheme: Traditional reconfigurable systems are *statically reconfigurable*, which means that the system is configured at the start of execution and remains unchanged for the duration of the application. In order to reconfigure a statically reconfigurable system, the system has to be halted while the reconfiguration is in progress and then restarted with the new configuration. *Dynamically reconfigurable* (*run-time reconfigurable*) systems, on the other hand, allow reconfiguration and execution to proceed at the same time. In this way dynamically reconfigurable systems permit the partial reconfiguration of certain logic blocks while others are performing computations.

Coupling: This refers to the degree of coupling with a host microprocessor. In a closely coupled system reconfigurable units are placed on the data path of the processor, acting as execution units. Loosely coupled systems act as a coprocessor. They are connected to a host computer system through channels or some special-purpose hardware.

Granularity of processing elements: This refers to the levels of manipulation of data. In *fine-grained* architectures, the basic programmed building block consists of a combinatorial network and a few flip-flops. These blocks are connected with a reconfigurable interconnection network. *Coarse-grained* architectures are primarily intended for the implementation of word-width data path circuits. Medium grained architectures can be considered as coarse grained architectures handling small word-width data.

The inclusion of reconfigurable hardware to a given system introduces significant advantages both from a market and from an implementation point of view. Currently equipment manufacturers move more and more towards solutions that can be upgraded in the field. This allows them to introduce first not fully completed products versions for time-to-market reasons and then extend products' lifetimes through firmware upgrades. The main reasons for this are:

- Need to conform to multiple or migrating international standards
- Emerging improvements and enhancements to standards
- Desire to add features and functionality to existing equipment

The presence of reconfigurable hardware in such a system allows low cost adaptivity since the same reconfigurable hardware may be shared among algorithms required for different operational conditions.

Finally reconfigurable hardware introduces the bug fixing capability for hardware systems in a software-like manner. In this way costly re-fabrications of VLSI components can be avoided.

Although the presence of reconfigurable hardware is advantageous in many cases significant overheads may be also introduced. These are mainly related to the time required for the reconfiguration and to the power consumed for reconfiguring (a part of) a system. Area implications are also introduced (memories storing configurations, circuit required to control the reconfiguration procedure). Furthermore existing system level methodologies need to be extended to cover issues related to the presence of reconfigurable hardware.

3. Reconfigurable technologies

Currently available (re)-configurable technologies can be classified in following major categories:

a) System level FPGAs: Currently available FPGAs offer sufficient density and speed to allow a complete design, from the microprocessor to all of its peripheral functions, in a single system-on-a-programmable chip. An example of system-level FPGA is the Xilinx Virtext-II Pro product family. Virtex-II Pro based family of FPGAs, embeds up to four RISC IBM PowerPC processors, using 32-bit CoreConnect on-chip bus. The Virtex-II Pro Architecture features up to 638K logic gates, 4 PowerPCs, 16 multi-gigabit transceivers and 3888Kbits BRAM. The PowerPC 405 RISC core provides over 300 MHz and 420 MIPS of performance, while consuming 0.9 mW/MHz of power. The Virtex-II Pro features 18Kbit block dual-port RAM, and 128-bit single-port or 64-bit dual-port distributed RAM. The device includes dedicated high-speed multipliers operating at 200 MHz pipelined, which support up to 18-bit signed or up to 17-bit unsigned representations, and they can be cascaded to support bigger numbers. Virtex family is dynamically reconfigurable comprising a fine grain architecture with granularity of 1-bit. The Virtex-II Pro architecture represents a typical SRAM based FPGA style device with regular arrays of CLBs surrounded by programmable input/output blocks.

b) Embedded reconfigurable cores/FPGAs: Embedded FPGA approaches are based on the idea of embedding bits of programmable logic into an ASIC in order to keep it (partly) programmable even after it comes out of the fab. An example of such reconfigurable technology is Actel's VariCore.

Actel's VariCore IP blocks are embedded, dynamically reprogrammable "soft hardware" cores designed for use in ASIC and SoC applications. Varicore is an architecture consisting of scaleable, configurable and partitionable programmable logic blocks from 2,500 to 40,000 ASIC gates for 0.18µ technology. VariCore EPGAs can be partitioned where needed throughout any ASIC or SoC design. Up to 73,728 bits of RAM can be available. PEGs are the primary logic blocks of VariCore EPGAs consisting of 2,500 ASIC gates. These PEG blocks are scaleable and configurable from a 2x1 EPGA of 5,000 ASIC gates up to an 8x8 EPGA of 160,000 ASIC gates (a maximum of 40,000 ASICgates in VariCore's 0.18 µm family). In addition, the .18µ EPGA family's 4x4 and 4x2 members offer eight optional, cascadable RAM modules with two aspect ratios of 1k*9 or 512*18. The first commercially available VariCore embedded programmable gate array (EPGA) blocks have been designed in 0.18 micron CMOS SRAM technology with 1.8V operating voltage for clock speeds up to 250 MHz.

Some power consumption figures include:

- 0.075 µW/Gate/MHz
- Typically 240 mW at 100 MHz and 80% utilization

c) Arrays of processing elements: Arrays of processing elements usually represents the combination of an instruction set processor (RISC) with reconfigurable fabric of coarse grain elements. MorphoSys [4] is a parallel system-on-chip which combines a RISC processor with an array of coarse-grain reconfigurable cells, with multiple context words, operating in SIMD fashion. It is primarily targeted for applications with inherent parallelism, high regularity, word-level granularity and computation intensive nature.

MorphoSys reconfigurable architecture is composed of a control processor, reconfigurable array (RA), data buffer, DMA controller, context memory and instruction/data cache.

In addition to typical RISC instructions, TinyRISC control processors ISA is augmented with specific instructions for controlling DMA and RA. The core processor executes sequential tasks of the application and controls data transfers between the programmable hardware and data memory.

The reconfigurable array consists of an 8×8 matrix of Reconfigurable Cells (RCs). An important feature of the RC Array is its three-layer interconnection network. The first layer connects the RCs in a two-dimensional mesh, allowing nearest neighbour data interchange. The second layer provides complete row and column connectivity and the third layer supports inter-quadrant connectivity. The RC is the basic programmable element in MorphoSys. Each RC comprises: an ALU-Multiplier, a shift unit, input multiplexers, a register file with four 16-bit registers and the context register.

The Context Memory stores the configuration program for the RC Array, the Frame Buffer stores the intermediate data.

The DMA controller performs data transfers between the Frame Buffer and the main memory. It is also responsible for loading contexts into the Context Memory. The TinyRISC core processor uses DMA instructions to specify the necessary data/context transfer parameters for the DMA controller.

MorphoSys is dynamically reconfigurable architecture. While the RC array is executing one of the 16 contexts, the other contexts can be reloaded into the context memory.

As can be seen, the different categories of dynamically reconfigurable technologies have very different characteristics and therefore, a unified model of them at the system-level is impossibility. One way of achieving accurate simulation results when doing design space exploration at system-level is to parameterise the configuration memory transfers at context switch and the delays associated with the reconfiguration process.

4. Related research and available tools

Since the dynamically reconfigurable systems potentially provide good tradeoff between performance and flexibility, there has been a lot of research going on in the field.A good review of reconfigurable computing has been done in [1]. The reconfigurable hardware, associated software and run-time reconfiguration technologies have been analysed in great detail.

Authors in [5] propose a complete compilation framework with a focus on task scheduling and context management problem for multicontext reconfigurable architecture. The algorithms are demonstrated on MorphoSys coarse-grained dynamically reconfigurable system. Authors present a task scheduling algorithm based on finding the optimal solution through the exploration of a reduced search space, with the aim to maximize data reuse and minimize context reloading. The problem of context loading management is tackled as two separate tasks: context selection and context allocation.

In [6] the authors review the high-level synthesis flows for dynamically reconfigurable systems, but conclude that there still is not mature algorithms or design flows available for truly supporting high-level synthesis of applications to dynamically reconfigurable hardware.

In [7] a methodology of doing technology independent simulation of dynamically reconfigurable hardware is presented using clock morphing where a virtual clock is distributed to different contexts of hardware, but the method is applicable at present time to RTL only. For system-level modeling authors of [8] presented a OCAPI-XL-based method where special processes called hardware scheduler automatically handles scheduling of contexts. However, the memory traffic associated to context switching is not modeled.

There are basically two types of approaches by the commercially supported flows: the tool oriented design flow or a language oriented design flow. As examples of tool oriented design flows are the N2C by CoWare [9] and VCC by Cadence [10]. Both of the design flows supported by these tools work well on traditional HW/SW solutions but since the refinement process of a design from unified and un-timed model towards RTL is tool-specific, the incorporation of new configurable domain is not possible without unconventional trickery. As example of language oriented design flow SystemC [11] can be used. Since the SystemC promotes the openness of the language and the standard, the addition of new domain can be made to the core language itself. However, a preferred method is to model the basic constructs required for modelling and simulation of reconfigurable hardware using basic constructs of the language and therefore preserving the compatibility with existing tools and designs.

5. Modeling methodology

Although there are several design and modeling methodologies available as described in Chapter 4, they have failed to reach widespread industrial and commercial adoption. This is due to several facts:

- The methodologies focusing on partitioning are not capable on handling IP introduced in different language or coding style.
- The partitioning algorithms assume that the application is implemented in single reconfigurable block and possibly RISC processor. In real life, there is usually need for more complex architectures.
- Co-simulation is not usually specified as goal of a methodology, which leaves much of the existing IP in a company unusable in system-level considerations.

For a co-design and modeling methodology to reach widespread support in the industry, following requirements must be satisfied:

- Use of existing code-base and IP must be simple.
- Co-simulation with existing models must be possible without modifications.
- The existing tools and methodologies must be applied to the design of dynamically reconfigurable devices, since there are both human and monetary investments in existing tools and design flows.
- The iterative design style must be supported, since a large portion of the designs are based on earlier versions of the same device.

5.1. The proposed design flow

The introduction of reconfigurable and more specifically, dynamically reconfigurable hardware brings new aspects especially to the system-level of the design flow. Instead of traditional hardware/software partitioning, the dynamically reconfigurable hardware is introduced. The effect dynamically reconfigurable hardware is separated from traditional ASIC hardware because of the temporal dimension introduced by reconfiguration process. The dynamically reconfigurable hardware also differs from software, because it can support the parallel execution and variable word lengths of hardware. So, it can be stated that the dynamically reconfigurable hardware brings a new dimension to the design space and the extra dimension sums up the complexities of both traditional hardware and software worlds.

Figure 3 describes a design flow adapted for support of dynamically reconfigurable hardware developed in the ADRIATIC project.

At the system-level part of the design flow (the bus-cycle accurate part) the adaptations of the design flow are not visible, but have considerable effects.

The system specification part is similar as without the use of dynamically reconfigurable hardware. The functionality of the system is implemented using a software language like C or C++. The executable specification can be used for several purposes:

- The test bench used in all phases of the design flow can be derived from the executable specification.
- The compiler tools and profiling information may be used to determine which parts of an application are most suitable for implementing with dynamically reconfigurable hardware. This is done in the partitioning phase of the design flow.
- The ability to implement executable specification validates that the design team has sufficient expertise on the application.

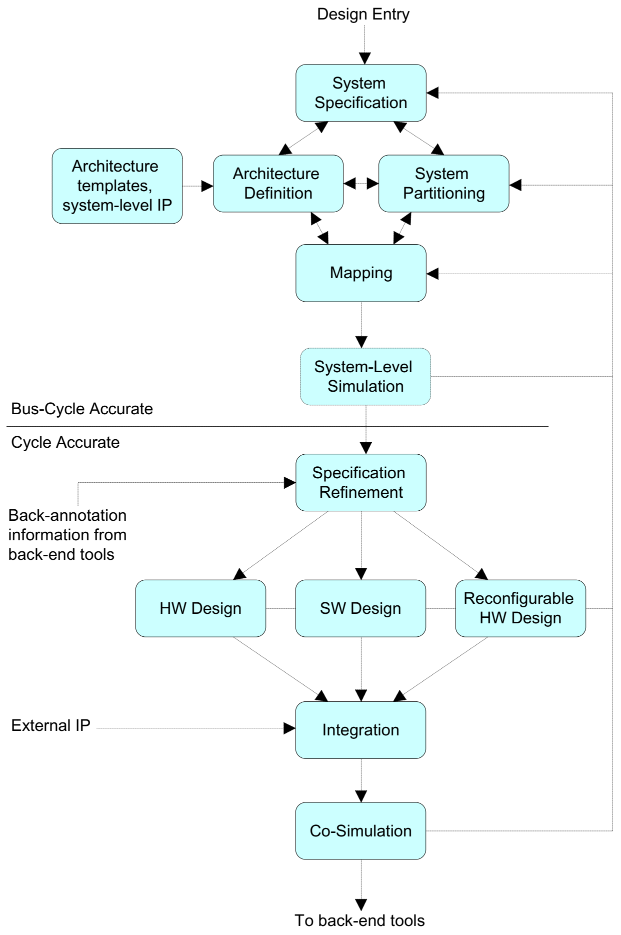

Figure 3. The ADRIATIC design flow.

The architecture of the device is defined partly in parallel and partly using the system specification as input. The initial architecture depends on several things. The company may have experience and tools for certain processor core or semiconductor technology, which restricts the design space. Also, a large part of all projects do not start from scratch, but they implement a more advanced version of an existing device. Therefore the initial architecture and the hardware/software partitioning is often given in the beginning of the system-level design. Also the reuse goals in each company mandate designers to reuse architectures and code modules developed in previous projects. The old models of an architecture are called architecture templates. In architectural design space, the dynamically reconfigurable hardware can be viewed as being a time-slice scheduled application specific hardware block.

In the partitioning phase, the functional blocks of executable specification are partitioned in to parts that will be implemented with software and parts implemented with hardware. In addition, the candidate blocks for implementation using dynamically reconfigurable blocks are identified. Although the detailed partitioning is not covered in this work and interested readers may refer to [5] for more information. However, there are some rules of a thumb that can be followed for identifying blocks implemented with reconfigurable hardware:

- If the application has several roughly same sized hardware accelerators that are not used in the same time or at their full capacity.
- If the application has some parts in which specification changes are foreseeable.
- If there are foreseeable plans for new generations of application, the parts that will change.

In the mapping phase of the system-level design flow, the functionality defined in executable specification is modified so that it simulates as accurately as possible the chosen implementation technology. Software parts may be compiled for getting some running time and memory usage statistics and hardware parts may be synthesized at high level to get estimates of gate counts and running speed. The functional blocks implemented with reconfigurable hardware are also modeled so that the effects of reconfiguration can be estimated. This is covered in detail in the next sub-chapter.

Finally in the system-level, some simulations are run to get information about the performance and resource usage of all architectural units in the device.

When considering the cycle accurate design of dynamically configurable hardware, the approach is somewhat simpler. The required tools are supplied by the chosen technology vendor. In the integration and co-simulation phases, there is a need to adapt the chosen reconfigurable technology to existing co-simulation flow. Also the verification process present in all phases must also take into account the implications of the reconfigurable technology. These aspects are however out of scope of this paper.

5.2. The modeling methodology for system-level

The modeling methodology presented here is based on SystemC [11]. For a description of the SystemC language, readers are urged to read [12].

The flow of the modeling methodology is shown in Figure 4. The modeling methodology takes a SystemC module as input and transforms the instance of the module and the hierarchical component that creates the instance to use a special component, a dynamically reconfigurable fabric (DRCF).

The main idea in the methodology is the following: When the functionality and the architecture are described in SystemC, *the methodology and associated tools provide means to test the effects of implementing some components in dynamically reconfigurable hardware*. This is achieved by automatically replacing candidate components with a special DRCF component, which implements a parameterized context scheduler and all functionality of the candidate components. The parameters include the modeling of memory bus traffic associated in context switching.

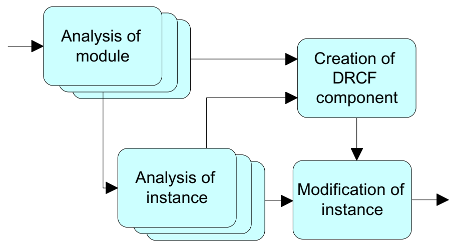

Figure 4. The modeling methodology phases.

In first phase, each module that is a candidate to implementation in reconfigurable hardware is analyzed. The used bus interface and the bus ports are analyzed so that the DRCF component can implement the same interfaces and ports.

After modules are analyzed, the methodology moves to analyze each instance of the modules in architecture. First the declaration of each instance is located and then the constructors are located and copied to a temporary database.

When all instances are analyzed, the DRCF component is created from a template. The ports and interfaces analyzed in the first phase are added to the DRCF template and then the component to be implemented in dynamically reconfigurable hardware is instantiated according to the declaration and constructor located in second phase. The template of the DRCF contains a context scheduler and instrumentation process and a multiplexer that routes data transfers to correct instances.

A simple example of what will be done to the SystemC models is shown next. The Code 1 listing of code shows a part of a simple hardware accelerator that was modeled and it is the candidate for implementation with dynamically reconfigurable hardware is our case.

Code 1. HW accelerator SystemC model:

```
class hwacc : public sc_module,
              public bus_slv_if
{
 public:
  sc_in_clk clk;
  sc_port<bus_mst_if> mst_port;
```

In the first phase of operation, the ports and interfaces of the module are analyzed. In this case, there module implements one interface *bus_slv_if* which is the slave interface of a bus and it has two ports *clk* and *mst_port*, which represent the clock input and master interface of a bus. Code 2 listing shows the definition of the *bus_slv_if*, the bus slave interface.

Code 2. Bus slave interface definition.

```
class bus_slv_if : public virtual
sc_interface
{
 public:
  virtual sc_uint<ADDW> get_low_add()=0;
  virtual sc_uint<ADDW> get_high_add()=0;
  virtual bool read(...)=0;
  virtual bool write(...)=0;
};
```

Now, we have the interface information of the module in form of interface methods and ports. In next phase, the instance of the module is analyzed. Code 3 shows the instantiation of the module in an hierarchical module called *top*.

Code 3. The top level SystemC model.

```
SC_MODULE(top){
  sc_in_clk clk;

  hwacc *hwa;
  bus   *system_bus;

  SC_CTOR(top) {
    system_bus = new bus("BUS");
    system_bus->clk(clk);

    hwa = new hwacc("HWA", HWA_START,
                    HWA_END);
    hwa ->clk(clk);
    hwa ->mst_port(*system_bus);
    system_bus->slv_port(*hwa);
```

From this listing, the declaration, constructor and the port and interface bindings are saved for later use. The declaration is the *hwacc *hwa*, the constructor is the line beginning with *hwa = new hwacc(* and the three lines under that show the port and interface bindings of the instance.

This hierarchical module is then updated to use the DRCF module instead of the hardware accelerator. The modified code is listed in Code 4.

Code 4. Modified top-level module.

```
SC_MODULE(top){
  sc_in_clk clk;

  drcf_own *drcf1;
  bus   *system_bus;

  SC_CTOR(top) {
    system_bus = new bus("BUS");
    system_bus->clk(clk);
    drcf1 = new drcf1("DRCF1");
    drcf1 ->clk(clk);
    drcf1 ->mst_port(*system_bus);
    system_bus->slv_port(*drcf1);
```

Notice that the declaration, the constructor and the binding lines are modified so that instead of the *hwa* instance a *drcf1* instance of a *drcf_own* is used.

In Code 5, the actual DRCF component created from a template is shown. In the code, all text that is in italics is the code that was inserted to the template.

Code 5. Creation of DRCF component.

```
class drcf_own : public sc_module
                 public bus_slv_if {
 public:
  sc_in_clk clk;
  sc_port<bus_mst_if> mst_port;

  hwacc *hwa;

  SC_HAS_PROCESS(drcf_own);

  void arb_and_instr();

  sc_uint<ADDW> get_low_add();
  sc_uint<ADDW> get_high_add();
  bool read(...);
  bool write(...);

  SC_CTOR(drcf_own) {
    SC_THREAD(arb_and_instr);
    sensitive_pos << clk;

    hwa = new hwacc("HWA", HWA_START,
                    HWA_END);
    hwa ->clk(clk);
    hwa ->mst_port(mst_port); } };
```

As can be seen, the interface and ports analyzed in the first phase are added to the component. Next, the declaration of the hardware accelerator is added as are the interface methods, constructor and the port bindings. What already was in the template is the *arb_and_instr()* method which handles the context scheduling and instrumentation.

In this example, a transformation process of a single module to be implemented was shown. In real life, a single context implemented with configurable hardware is not dynamically reconfigurable, since there is no need in changing the context. To fully exploit the automatic context scheduling provided, several models are transformed in to a same DRCF.

5.3. The context scheduler

When several modules are implemented in same reconfigurable hardware, context switches happen. The context switch does not only create delay to the activities because of the reconfiguration, but it also creates bus transformations, which may harm the total performance of the system. The context switches and the bus transfers should be automatically modeled for quick and accurate design space exploration at system-level. When considering the implementation technologies such as described in Chapter 3, the need for parameters arise. In our first specification of the modeling methodology, there are parameters for each context available for designer:

1. The memory address, where the context is allocated.
2. The size of the context.
3. Delays associated with the re-configuration process (in addition to the delays of memory transfers).

In the future, other parameter, such as dealing with partial reconfiguration or power consumption may be devised.

The behavior of the context scheduler is the follwing:

1. When an interface method is called, the context scheduler checks to which component the interface method call was targeted to.
2. If the interface method call was targeted to the active context, the interface method call is forwarded directly.
3. If the interface method call was targeted to a context which is not active, the context switch is activated.
4. During context switch, the interface method call is suspended until the arbitration and instrumentation process has generated proper data reads in to the memory space that holds the required context.
5. The scheduler will keep track of active time of each context as well as the time that the DRCF is in reconfiguring itself.

This process automatically models the context switching and the memory bus traffic. In addition, this methodology may be used to measure the effects of different memory organizations or implementation to the total system performance.

5.4. Current limitations of the methodology

There are however some limitations in the methodology which may require the designer to modify the design. These restrictions are SystemC specific and therefore the implementation of this methodology in other languages may not contain these limitations.

1. All models that are transformed in to a DRCF implementation must be on same level of hierarchy and instantiated in the same component.
2. All implemented interfaces must contain two interface methods that are used to finding out the memory space of a single component. In our example these methods were the *get_low_add()* and *get_high_add()*. This seems to be a very common way of implementing interfaces in system-level models in SystemC 2.0.
3. The interface methods must be non-blocking or must support split transactions if the context memory bus is

the same as the interface bus of the components. If this is not the case, a data transfer to a component in DRCF would block the bus until the transfer is completed and the DRCF could not load a new context, since the bus is already blocked. This results in deadlock of the bus.

5.5. The effect of implementation technologies on system-level models

When assessing the parameters of the different kinds of reconfigurable technologies described in Chapter 3, following observations can be made: When choosing the reconfigurable technology, there are three major issues that need to be modeled for getting reliable information about the trade-off issues between area, speed and total cost. First one is the processing speed of a functional block, second is the required resources needed for largest context and third is the delays and memory consumption caused by the reconfiguration. All these parameters are technology dependent so the best that a practical modeling methodology at system-level can do is to define a set of parameters to model different implementation technologies. Parameterized modeling enable automatic design space exploration with best possible accuracy. This methodology tries to ease the consistent estimation of the effects of a specific implementation technology at system-level.

6. Discussion and further work

The proposed modeling methodology is far from complete. In the presented small example and other cases, the transformations are done by hand according to specification. Part of the work in the project ADRIATIC will be directed to implementation of the tools that are required for this methodology to be fully automatic. Also some research will be done on finding the correct parameters at system-level to reach good accuracy when compared to actual implementation in some selected target reconfigurable hardware.

Also the analysis methods of the system-specification need to be investigated so that there could be tool-based input to designer hinting which parts of the application are candidates to implementation in dynamically reconfigurable hardware. Currently, the methodology assumes that the designer has the initial idea of the partitioning and he can verify the system performance via quick and automated design space exploration.

However, there seems to be a need for a practical system-level methodology addressing the dynamically reconfigurable hardware, which does not rely on tools and methodologies that are not really used in the industry. The SystemC 2.0 language is a good candidate to a industry standard system-level design language, since there are already many companies using it and since there are modeling, co-simulation, synthesis etc. tools available.

7. Conclusions

A methodology for system-level modeling of dynamically reconfigurable hardware using SystemC was shown with a simple example of how this is done. The modeling methodology may also be used for making quick design space exploration when considering which functional blocks of an application will be implemented with dynamically reconfigurable hardware. There is still work to do on the tools of the methodology and further investigations of the accuracy of the results when comparing them to actual implementation in specific real reconfigurable hardware.

Acknowledgements

This work is supported by EU through the IST-2000-30049 ADRIATIC project. The authors gratefully acknowledge the discussions with colleagues in project ADRIATIC and others.

References

[1] Compton K., Hauck S., “Reconfigurable Computing: A Survey of Systems and Software”, ACM Computing Surveys, June 2002, pp. 171-210.

[2] DeHon A., Wawrzynek J., “Reconfigurable Computing: What, Why, and Implications for Design Automation”, Proceedings of 36th DAC, June 1999.

[3] B. Brodersen, “Wireless System-on-a-Chip Design”, http://bwrc.eecs.berkeley.edu/

[4] Hartej S, et al. "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE transactions on computers. May 2000, pp. 465-481.

[5] Maestre R., et al., "A framework for reconfigurable computing: task scheduling and context management", IEEE Transactions on VLSI Systems, December 2001, pp. 858-873.

[6] Zhang X., Ng K.W., "A review of high-level synthesis for dynamically reconfigurable FPGAs", Microprocessors and Microsystems, August 2000, pp. 199-211.

[7] Vasilko M., Cabanis D., “Improving Simulation Accuracy in Design Methodologies for Dynamically Reconfigurable Logic Systems”, Proc. of FCCM, 1999, pp. 123 -133.

[8] Rissa T., Vasilko M., Niittylahti J., “System-Level Modeling and Implementation Technique for Run-Time Reconfigurable Systems”, Proc. of FCCM, April 2002.

[9] http://www.coware.com/cowareN2C.html

[10] http://www.cadence.com/products/vcc.html

[11] Panda P.R., "SystemC - a modeling platform supporting multiple design abstractions", Proceedings of the 14th ISSS, 2001 pp. 75 -80.

[12] The Functional Specification for SystemC 2.0, http://www.systemc.org/