In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen.* by Christian Haubelt, Jürgen Teich (Eds.). GI/ITG/GMM-Workshop, Shaker Verlag, Aachen, March 5-7, 2007

System Level Modeling and Performance Simulation for Dynamic Reconfigurable Computing Systems in SystemC

Martin Streubühr, Carsten Riedel, Christian Haubelt, Jürgen Teich
Hardware–Software–Co–Design, Department of Computer Science 12
University of Erlangen–Nuremberg, D–91058 Erlangen, Germany
{streubuehr, riedel, haubelt, teich}@codesign.informatik.uni-erlangen.de

Designing reconfigurable computing systems is a challenging task; especially forecasting the benefits from using reconfigurable architectures at design time is hard. As a solution, we present a simulation–based framework for performance evaluation of partitioned reconfigurable computing systems. With this framework, the designer is enabled to assess the impact of different scheduling and reconfiguration strategies at a high level of abstraction with marginal simulation overhead and minimal source code modifications. We will show the benefits of our approach by comparing our performance simulation to a pure functional simulation.

1 Introduction

When comparing different kinds of reconfigurable hardware [4, 14] with different levels of granularity, there is one common feature: The ability of reusing a piece of hardware for different missions. By loading configuration bit streams, FPGAs provide flexibility comparable to general–purpose processors while having an impressive performance. A newer trend in reconfigurable computing systems is to exploit advantages of dynamic reconfiguration. That implies swapping different temporally needed FPGA–accelerated functionalities during run time of a system. In this context, many new problems like scheduling, mapping, partitioning, etc. arise.

The *Virtual Processing Components* (VPC) framework [13] allows for simulation–based performance evaluation of applications mapped to heterogeneous multi–processor architectures at an early stage in design. For this purpose, the VPC framework uses the SystemC language for representing the candidate architectures at system level. In this work, we present the extensions towards modeling and simulation of reconfigurable computing systems. By using a system level model of partitioned reconfigurable architectures [15] our approach abstracts from device and architecture dependent implementation details, leading to the concept of *virtual* components. The Erlangen Slot Machine (ESM) [3] is one example for a typical partitioned reconfigurable architecture.

The aim of using the VPC framework is answering the following questions in an early design stage: Which performance (e.g. throughput, latency) provides my application using different architectures? Can my application benefit from using reconfigurable devices? Which reconfiguration strategies, like partitioning, allocation, and binding fits best to my application?

The remainder of this document is structured as follows: In Section 2, we review existing works on simulation of reconfigurable computing systems. Section 3 describes the modeling and Section 4 presents the performance simulation of partitioned reconfigurable computing systems. We give an example in Section 5 and a conclusion in Section 6.

2 Related Work

In [12] a modeling technique for reconfigurable systems is proposed based on *OCAPI–xl* — a modeling language similar to SystemC. A set of processes is used to model a *run time reconfigurable* process which is supervised by a *HardWare Scheduler* (HWS). The HWS coordinates the exclusive execution of the pooled processes and activates or stops them. Additional reconfiguration latency can be accounted by the HWS.

Noguera and Badia [9] present a modeling methodology for reconfigurable architectures based on discrete event systems. In an object–oriented manner discrete event classes and objects form a reconfigurable system, which introduces all necessary modeling capabilities to cover run time reconfiguration. Traditional hardware/software scheduling as well as dynamic reconfigurable logic scheduling is considered in this approach. The authors point out that both scheduling aspects should be handled as related entity. Dynamic scheduling strategies are taken into account by proposing a novel scheduling methodology based on the presented event system.

In [5] a Co–Simulation environment is unveiled enabling performance estimation for a dedicated architecture which couples a CPU together with a coarse–grained, multi–context reconfigurable unit (RU). The approach addresses the impact of multi–context RUs and focuses on the coupling of a multi–context device and a controlling CPU. Performance evaluation is based on a cycle–accurate simulation including execution and reconfiguration times. The CPU behavior is modeled by an extension of the *SimpleScalar* CPU simulator [1], while the RU itself is implemented in VHDL. Both, the CPU and the RU, cooperate through a dedicated coprocessor interface. The main focus of this approach is the investigation of design trade–offs for multi–context architectures.

The authors of [10, 11] propose a method for modeling dynamically reconfigurable hardware in SystemC. The presented approach transforms a specification — given in SystemC — to model reconfigurable hardware. For this purpose a special component is introduced called *dynamically reconfigurable fabric* (DRCF) *component*. This component includes a *configuration scheduler*, an *input splitter*, and an associated *configuration memory*. After analyzing the SystemC modules the DRCF code is constructed using a template. During this process, the reconfigurable functionality is identified and integrated into different configurations within a DRCF. This source code modification

enables modeling of context switching and bus traffic.

The system–level framework *Perfecto* for rapid explorations of different reconfiguration alternatives and performance evaluation is presented in [8]. A system architecture model is outlined containing a *DRCL* specified in SystemC. Each configuration is associated with a size given by an amount of *Slices*, the basic configuration unit in the framework. A *DRCL* provides a certain amount of *Slices*, and dynamic reconfiguration is modeled by placing configurations into the limited area of a *DRCL*, similar to memory management. Through additional information, e.g. task execution time or configuration time, *Perfecto* is able to estimate aspects like total execution time and overall slice usage.

Unlike other approaches, our VPC framework clearly separates application behavior and architecture. By using a flexible mapping mechanism, we are able to evaluate different bindings between applications and architectures, as well as different reconfiguration and task scheduling strategies, without any further source code modification. This allows for reusing both, the application and the architecture templates.

3 Modeling Reconfigurable Computing Systems

In this section, we develop our system level model for reconfigurable computing systems. The resulting model is used later on to set up our performance simulation.

In [6] a graph–based approach is presented for modeling the hardware/software partitioning problem for architectures including reconfigurable hardware resources. Each partitioning problem instance consists of a *process graph* g_p and the *architecture graph* g_a. The explicit separation between *process graph* and *architecture graph* fits perfectly to our strict separation between application behavior and architecture and provides an ideal model for our simulation–based approach.

Definition 1 (Process Graph [6]) *A* process graph $g_\mathrm{p} = (V_\mathrm{p}, E_\mathrm{p})$ *consists of a finite set of vertices* V_p *and a finite set of directed edges* $E_\mathrm{p} \subseteq V_\mathrm{p} \times V_\mathrm{p}$ *where vertices model processes and edges represent communication links between processes.*

In this work, we assume that the *process graph* can be extracted from a SystemC description. Nodes $v \in V_\mathrm{p}$ in a *process graph* represent tasks implemented in SystemC, as given later in section 4. An edge $e \in E_\mathrm{p}$ between vertices represents directed communication between those tasks (e.g. FIFO channels).

Definition 2 (Architecture Graph [6]) *An* architecture graph $g_\mathrm{a} = (V_\mathrm{a}, E_\mathrm{a})$ *consists of a finite set of vertices* V_a *and a finite set of directed edges* $E_\mathrm{a} \subseteq V_\mathrm{a} \times V_\mathrm{a}$. *The vertices* $v \in V_\mathrm{a}$ *of the* architecture graph g_a *correspond to hardware modules like CPUs, memories, FPGAs, etc. The edges* $e \in E_\mathrm{a} \subseteq V_\mathrm{a} \times V_\mathrm{a}$ *of the* architecture graph g_a *correspond to connections between hardware modules modeling the topology of the architecture. Furthermore, vertices* $v \in V_\mathrm{a}$ *in the* architecture graph *may be refined by subgraphs* $\widetilde{g} \in v.G$ *associated with* v.

In an architecture graph, a vertex $v \in V_\mathrm{a}$ may contain a set of subgraphs $G' \subseteq v.G$. Each subgraph $g \in G'$ can be considered as a hierarchical refinement of vertex v. All refinements $\widetilde{g} \in v.G$ within node v are *mutually exclusive*, i.e., they cannot be used for refinement at the very same instance of time. A vertex that can be refined is called

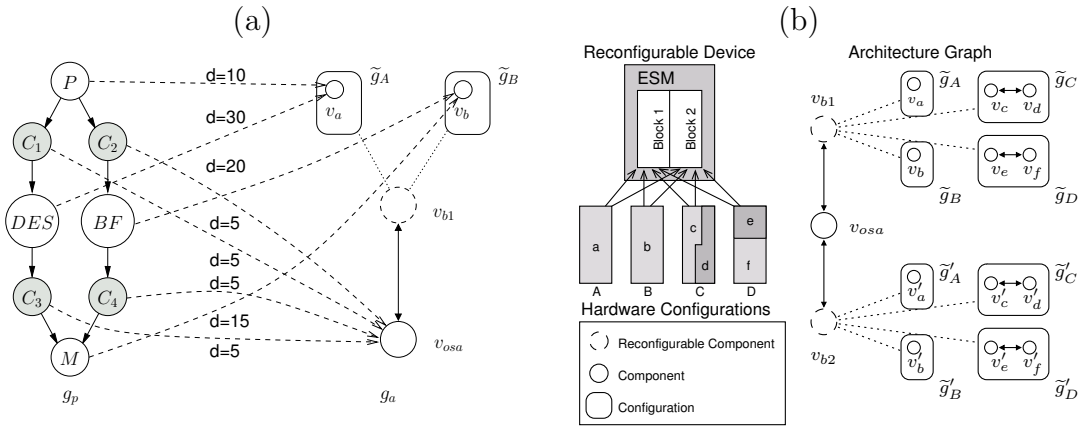

Figure 1: (a) The *process graph* g_p is mapped to a hardware reconfigurable architecture g_a. The resource v_{b1} is refined by two different configurations $\widetilde{g}_A, \widetilde{g}_B$. (b) Partitioned reconfigurable devices offer blocks for hardware module placement [15]. Each block is represented by a node $v \in V_\mathrm{a}$. Each hardware configuration for a block v is modeled through refinements $\widetilde{g} \in v.G$ of node v.

Reconfigurable Component, otherwise ($v.G = \emptyset$) it is called *Component*. Refinements of a *Reconfigurable Component* are named *Configurations*.

The ability of implementing a process v_p using a resource v_a is modeled by a *mapping edge* $m = (v_p, v_a) \in E_\mathrm{m}$ where E_m is called set of mapping edges and is depicted by dashed arcs in Figure 1 (a). Several property values can be associated to the mapping edges such as execution delays, area, monetary cost, etc.

Process graph, *architecture graph* and *mapping edges* together form a *specification graph* [2]. As a *specification graph* typically contains several *mapping edges* per process, the resulting partitioning problem can be solved by: (i) *Allocation*: Selecting a subset of the vertices and subgraphs of the given architecture. (ii) *Binding*: Choosing a subset of the *mapping edges* so that only one mapping rests for each process. (iii) *Scheduling*: Finding a valid schedule for a given allocation and binding.

An *architecture graph* can be used to model a partitioned reconfigurable device as depicted in Figure 1 (b): Each block that can be reconfigured by different configurations is modeled as a vertex $v \in V_\mathrm{a}$ which contains a set of subgraphs $G' \subseteq v.G$. Each of those hierarchical refinements represents a single configuration that can be loaded into that block. If a configuration can be relocated to different blocks, it is necessary to model a subgraph associated to each vertex on its own. For example, the subgraphs $\widetilde{g}_A$ and $\widetilde{g}'_A$ in Figure 1 (b) represent the same configuration which can be loaded onto different blocks depicted by vertices v_{b1} and v_{b2}.

4 Performance Simulation

Previous approaches [2, 6] could construct a static schedule for an implementation by limiting their methodology to a homogeneous data flow model of computation as semantics for the *process graph*. Nevertheless, these approaches are neither extensible towards

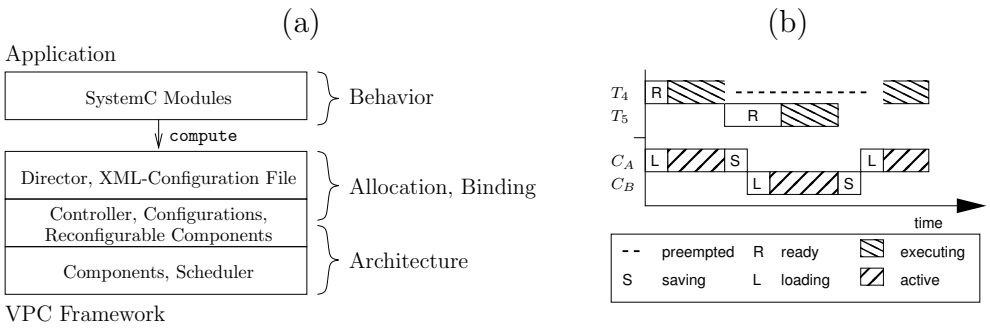

Figure 2: (a) An application is given by a set of communicating tasks given as SystemC modules, and the architecture is modeled by the VPC framework. The framework provides the `compute` interface, allowing the application to access performance simulation. Binding between application and architecture is managed within our framework. (b) Gantt Chart representing temporal behavior of configuration switch including context saving.

more general models of computation nor do they consider arbitrary scheduling and reconfiguration strategies. Our approach overcomes those limitations by: (i) simulate the application given as communicating SystemC modules, (ii) modeling of reconfigurable architectures and (iii) simulate online allocation, binding, and scheduling strategies. This simulation–based methodology enables performance evaluation of applications given in SystemC using partitioned reconfigurable architectures.

Figure 2 (a) gives an overview of our approach: In conformity with the *specification graph* our simulation methodology separates between application, architecture, and mappings. The application is given by a set of sequential tasks implemented in SystemC, forming a multi–tasking system. The VPC framework provides *Virtual Processing Components* and *Reconfigurable Components* for modeling a reconfigurable multi–processor architecture. The *Director* is used for mapping the application onto the architecture and therefore provides a special `compute` function called from the SystemC tasks. In the following we will discuss some details of our approach.

4.1 Application

An application consists of a set of communicating tasks given in SystemC (the *process graph*). Each task is represented in SystemC by an `SC_MODULE` containing an `SC_THREAD`. As the SystemC reference implementation comes together with a simulation kernel, such applications can be functionally simulated. Each task performs computations if it is executed, and possibly communicates with other task by exchanging data. Later, in a real implementation, each task needs some time for computation and communication, and thus has an impact on the performance of a system.

As SystemC contains a model of time we possibly could insert `wait` function calls with respect to the target architecture. But this implies the modification of the SystemC source code for each candidate architecture. Using our Virtual Processing Components framework, a more sophisticated way is introduced by the `compute` function call: In-

stead of annotating execution delays using `wait`, we use `compute` as the interface to our framework. The VPC framework simulates the execution delays by blocking the `compute` function. Once `compute` returns, the entire execution time is expired and the task may be started for another computation.

4.2 Architecture

The VPC framework forms the architecture by instantiating a set of *Components* and *Reconfigurable Components*. *Components* and *Reconfigurable Components* are designed as `SC_MODULEs` permitting to simulate execution delays using the model of time provided by SystemC. At simulation startup the entire specification is constructed by loading an XML formated configuration file and initializing the given architecture and mapping. The so called *Director* provides the `compute` function to be called from application tasks. Each call of this function is forwarded to a *Component* or a *Reconfigurable Component*, according to the binding for simulating the execution delay.

A *Component* represents a hardware resource like a CPU, a dedicated hardware module, etc. Being associated with a *Scheduler*, a *Component* is able to simulate resource contention if several tasks call `compute` in parallel. As already presented in [13], we provide different preemptive and non–preemptive scheduling strategies.

The contribution of this work is the modeling and simulation of reconfigurable computing systems. A *Reconfigurable Component* contains several mutually exclusive *Configurations* and is associated with a *Controller* for managing the *Configurations*. That means, to bind tasks to *Configurations* and to allocate those *Configurations*. Binding means the selection of a *mapping edge* between a task and a *Component*. If this *Component* belongs to a *Configuration* of a *Reconfigurable Component*, then it is a prerequisite for the task execution that this particular *Configuration* is allocated. Hence, loading or unloading configurations has to be performed if it is necessary. To reflect the time for configuration switching each *Configuration* $\widetilde{g}_i \in v.G$ has a *context restoration time* $load(\widetilde{g}_i)$ and a *context saving time* $store(\widetilde{g}_i)$.

Figure 2 (b) depicts an example using a preemptive context–saving allocation strategy. Two tasks T_4 and T_5 are mapped onto two mutually exclusive *Configurations* C_A and C_B, respectively. Task T_4 is executed first, but its *Configuration* is preempted by task T_5 needing another *Configuration* for execution. Both, the execution times and the reconfiguration overhead, affects the execution delays for the tasks T_4 and T_5.

As mentioned earlier, each *Reconfigurable Component* is supervised by a *Controller* to manage the allocation and binding strategy. Currently our *Controllers* support the following allocation algorithms: *Priority Allocation*, *Round Robin*, *Earliest Deadline First*, and *First Come First Served*. Additionally we provide the following binding strategies: *Round Robin*, *Priority Binding*, and *Avoid Reconfiguration*. *Least Currently Bound* (*LCB*) and *Least Frequently Used* (*LFU*) are examples of binding policies using the *Priority Binding* algorithm with dynamic priorities. Further strategies for allocation and binding can easily be integrated by implementing new *Controllers*.

The VPC framework permits for nesting of several hierarchy layers of *Reconfigurable Components*. This way we can cover the special case of loading a big size module in

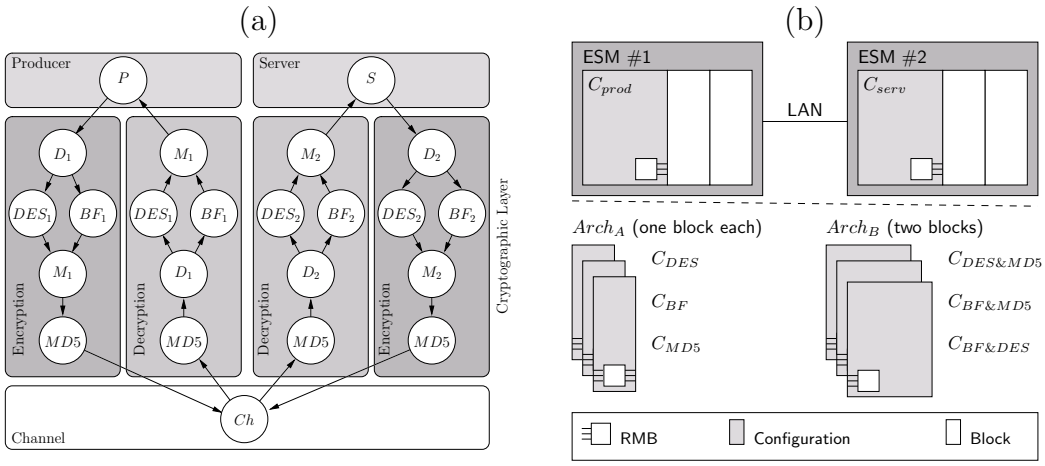

Figure 3: (a) The example application: A producer sends computation requests to a server and the results are returned to the producer. The RPC–like communication over the channel is secured using a cryptographic layer. (b) The underlying Architecture consists of two ESM boards [3]. Two different configurations, $Arch_A$ and $Arch_B$ can be loaded into one or two reconfigurable blocks, respectively.

several block at the same time: We use one *Reconfigurable Component* to represent a grouping of blocks. That means, it can allocate a large module or a fixed number of *Reconfigurable Components* for loading smaller blocks.

4.3 Evaluation Flow

From an application engineers point of view the usage of our framework is as follows: (i) Implement an application in SystemC and augment each task with an arbitrary number of `compute` statements in order to represent execution or communication delays. (ii) Link the application with our VPC framework. (iii) Design a configuration file to define the architecture and the mappings for the performance evaluation. (iv) Run simulation to evaluate the performance of the obtained system. (v) For comparison mutate the configuration file to simulate different architecture designs and mappings as well as different reconfiguration and scheduling strategies.

The viewpoint of a system engineer may be to design and test new reconfiguration and scheduling strategies for comparison with existing algorithms. In this case, the developer needs to implement the *Controller* and *Scheduler* interfaces.

5 Experimental Results

This section outlines experimental results of our approach by using the encryption and decryption application depicted in Figure 3 (a). The dedicated communication nodes at each dependency edge are not shown for simplicity. A producer P sends requests to a server S. After execution the results are send back to P. Assuming that the exchanged

Table 1: The four different mappings ($Arch_A$, $Arch_B$, Hardware, Software) between example application and reconfigurable architecture and the according core execution times. The configurations $Arch_A$, $Arch_B$, and Hardware make use of the same execution times.

task	**Arch$_A$**	**Arch$_B$**	**Hardware**	delay	**Software**	delay
P	C_{prod}	C_{prod}	C_{prod}	2 ns	C_{P_1}	2 ns
S	C_{serv}	C_{serv}	C_{serv}	2 ns	C_{P_2}	2 ns
D_1	C_{prod}	C_{prod}	C_{prod}	1 ns	C_{P_1}	1 ns
D_2	C_{serv}	C_{serv}	C_{serv}	1 ns	C_{P_2}	1 ns
M_1	C_{prod}	C_{prod}	C_{prod}	1 ns	C_{P_1}	1 ns
M_2	C_{serv}	C_{serv}	C_{serv}	1 ns	C_{P_2}	1 ns
Ch	C_{LAN}	C_{LAN}	C_{LAN}	35 ns	C_{LAN}	35 ns
BF_x	C_{BF}	C_{BF+DES}, C_{BF+MD5}	C_{BF}	242 ns	C_{P_x}	7.28 ms
DES_x	C_{DES}	C_{BF+DES}, $C_{DES+MD5}$	C_{DES}	91 ns	C_{P_x}	352.44 ms
$MD5_x$	C_{MD5}	C_{BF+MD5}, $C_{DES+MD5}$	C_{MD5}	687 ns	C_{P_x}	12.13 ms

data should kept secret, the communication is encrypted using either Data Encryption Standard (DES) or Blowfish (BF), and additionally each packet is signed using an MD5 algorithm ($MD5$). As there is a bidirectional communication the same en– and decryptions have to be performed backward to P. The communication between P and S is done via the communication channel Ch.

The case study focuses on the evaluation of different reconfigurable architectures using the Erlangen Slot Machine (ESM) [3] as target architecture. We assume the communication between modules is done via a reconfigurable multiple bus (RMB) [3]. Two different architecture arrangements ($Arch_A$, $Arch_B$) have been considered as depicted in Figure 3 (b). Both arrangements consist of a static block (C_{serv}, C_{prod}) providing modules for producer or server. The first arrangement provides two blocks for loading encryption modules either performing en–/decryption (C_{DES}, C_{BF}) or signing of packets (C_{MD5}). The second arrangement has only one large block using combined modules ($C_{DES+MD5}$, C_{BF+MD5}, C_{BF+DES}). Note that it is not possible to load the required functionality completely into the reconfigurable blocks. Table 1 specifies the different mapping possibilities for each architecture used as well as the assumed core execution time for the application tasks depending on the mapping to either software or hardware. To estimate the software execution times we run functional simulation on a workstation and scaled the resulting execution time to an assumed clock rate of 200MHz for an embedded processor. The hardware execution times are taken from data–sheets of corresponding encryption cores from Helion Technology [7].

Table 2 shows the determined execution times in our case study. Three different combinations of allocation and binding strategies have been evaluated. (i) A simple *FCFS* configuration allocation combined with a *Round Robin* binding strategy. (ii) *FCFS* allocation combined with a *Least Currently Bound* (LCB) binding strategy. (iii) A strategy based on *Round Robin* allocation and *Avoid Reconfiguration* binding. For comparison, we also simulated a software architecture and a hardware–only architecture, the results are also given in Table 2. The software architecture is assumed to be an embedded processor at 200MHz clock rate running the cryptographic modules. As hardware–only

Table 2: Average end to end latency per executed request for the worst case (WC) and the both best case (BC) scenarios.

latency [ms]	$Arch_A$			$Arch_B$			Hardware	Software
Allocation	FCFS	FCFS	RR	FCFS	FCFS	RR		
Binding	RR	LCB	ARB	RR	LCB	ARB		
BC Blowfish	62.61	11.03	0.03	234.81	62.75	0.01	0.01	5487.67
BC DES	62.73	10.99	0.03	236.02	63.75	0.01	0.01	293.26
WC	82.39	7.91	0.07	268.26	227.96	0.14	0.01	4066.80

Table 3: Average execution times for simulation the given architectures of 100 simulation runs using 550 requests at the producer. The overhead of profiling the architecture using the VPC framework is given in percent.

runtime	$Arch_A$			$Arch_B$			Hardware	Software	w/o VPC
Allocation	FCFS	FCFS	RR	FCFS	FCFS	RR			
Binding	RR	LCB	ARB	RR	LCB	ARB			
BC BF	15.1s	13.9s	14.8s	14.9s	13.9s	14.9s	11.8s	11.4s	8.2s
	85.0%	70.6%	80.5%	81.7%	69.9%	82.0%	44.0%	40.0%	0%
BC DES	94.9s	93.5s	94.4s	94.5s	93.5s	94.4s	91.3s	90.8s	86.8s
	9.3%	7.7%	8.7%	8.9%	7.7%	8.7%	5.1%	4.6%	0%
WC	55.3s	53.6s	54.7s	55.0s	54.4s	55.1s	51.8s	51.3s	47.9s
	15.3%	11.8%	14.2%	14.7%	13.4%	14.9%	8.1%	7.1%	0%

architecture, we assumed a larger ESM, so that all modules can fit in, and no reconfiguration is necessary.

For the evaluation three different scenarios have been used, a homogenous communication just using DES and another only using BF encryption to represent best case (BC) scenarios. The third set consists of alternating encrypted packets to represent a worst case (WC) scenario.

We have measured the average execution time for 100 simulation runs. Each assumes 550 requests generated by the producer. In Table 3 we show the resulting average execution times per simulation run. Additionally, Table 3 shows the overhead of the simulation with resource profiling compared to a pure functional simulation run. The maximum overhead in this scenarios is less than twice of the functional simulation time. Note that the overhead caused by using the VPC framework hardly depends on the complexity of the application. For example, all best case scenarios using Blowfish encryption produce a larger overhead but a smaller runtime compared to DES encryption. This is due to the fact that, Blowfish encryption is faster then DES encryption in our implementation.

6 Conclusions

In this paper, we have presented an extension of the Virtual Processing Components framework towards dynamically reconfigurable architectures. The VPC framework is used to represent architectures and mappings, allowing for performance evaluation, and also provides mechanisms for developing reconfiguration and scheduling algorithms using SystemC. Using a high level of abstraction, our approach comes with marginal simulation overhead and minimal source code modifications.

References

[1] T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] T. Blickle, J. Teich, and L. Thiele. System-Level Synthesis Using Evolutionary Algorithms. In Rajesh Gupta, editor, *Design Automation for Embedded Systems*, 3, pages 23–62. Kluwer Academic Publishers, Boston, January 1998.

[3] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. Increasing the Flexibility in FPGA-Based Reconfigurable Platforms: The Erlangen Slot Machine. In *IEEE 2005 Conference on Field-Programmable Technology*, pages 37–42, Singapore, Singapore, December 2005.

[4] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[5] R. Enzler, C. Plessl, and M. Platzner. Co-Simulation of a Hybrid Multi-Context Architecture. In *Engineering of Reconfigurable Systems and Algorithms*, pages 174–180, 2003.

[6] C. Haubelt, S. Otto, C. Grabbe, and J. Teich. A System-Level Approach to Hardware Reconfigurable Systems. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 298–301, Shanghai, China, January 2005.

[7] HELION Technology Limited. http://www.heliontech.com, June 2006.

[8] C.-F. Liao and P.-A. Hsiung. A SystemC-based Performance Evaluation Framework for Dynamically Reconfigurable SoC. In *Proceedings of the VLSI Design / CAD Symposium*, pages 444–447, August 2005.

[9] J. Noguera and R. M. Badia. System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 73–83, New York, NY, USA, 2003.

[10] A. Pelkonen, K. Masselos, and M. Cupák. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 174–181, Washington, DC, USA, April 2003.

[11] Y. Qu, K. Tiensyrja, and J.-P. Soininen. SystemC-based Design Methodology for Reconfigurable System-on-Chip. In *Proceedings of the 8th Euromicro Conference on Digital System Design*, pages 364–371, Los Alamitos, CA, USA, 2005.

[12] T. Rissa, M. Vasilko, and J. Niittylahti. System-Level Modelling and Implementation Technique for Run-Time Reconfigurable Systems. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 295–296, Los Alamitos, CA, USA, 2002.

[13] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf. Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. In *Proceedings of Design, Automation and Test in Europe*, pages 480–481, Munich, Germany, March 2006.

[14] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings – Computers and Digital Techniques*, 152(2):193–207, March 2005.

[15] H. Walder and M. Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of Design, Automation and Test in Europe*, pages 290–295, March 2003.