Available online at www.sciencedirect.com

SCIENCE DIRECT®

Microprocessors and Microsystems 29 (2005) 63–73

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

System-level performance evaluation of reconfigurable processors

R. Enzler[a,*], C. Plessl[b], M. Platzner[b]

[a]*Electronics Laboratory, Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, ETH Zentrum, CH-8092 Zurich, Switzerland*
[b]*Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, ETH Zentrum, CH-8092 Zurich, Switzerland*

Received 20 October 2003; revised 8 March 2004; accepted 1 June 2004
Available online 20 July 2004

Abstract

Reconfigurable architectures that tightly integrate a standard CPU core with a field-programmable hardware structure have recently been receiving increased attention. The design of such a hybrid reconfigurable processor involves a multitude of design decisions regarding the field-programmable structure as well as its system integration with the CPU core. Determining the impact of these design decisions on the overall system performance is a challenging task. In this paper, we first present a framework for the cycle-accurate performance evaluation of hybrid reconfigurable processors on the system level. Then, we discuss a reconfigurable processor for data-streaming applications, which attaches a coarse-grained reconfigurable unit to the coprocessor interface of a standard embedded CPU core. By means of a case study we evaluate the system-level impact of certain design features for the reconfigurable unit, such as multiple contexts, register replication, and hardware context scheduling. The results illustrate that a system-level evaluation framework is of paramount importance for studying the architectural trade-offs and optimizing design parameters for reconfigurable processors.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Field-programmable gate arrays; Reconfigurable computing; Hybrid reconfigurable processors; Co-simulation; VHDL

1. Introduction

It has been shown that field-programmable devices can achieve significant speedups and power savings for important application kernels compared to general-purpose CPUs [1–3]. However, mapping complete applications rather than kernels to a field-programmable device is difficult and often inefficient. The ideal platform combines a field-programmable device with a general-purpose CPU. Consequently, devices that integrate a fixed general-purpose CPU with a field-programmable unit have received increasing attention in the last years. We denote such devices as *hybrid reconfigurable processors*.

The design of a hybrid reconfigurable processor involves a multitude of design decisions related to finding an optimal architecture of the reconfigurable unit (RU) and a suitable system integration of the CPU and the RU [3–5]. The design decisions for the RU architecture, i.e. type of configurable logic blocks, routing architecture, and configuration architecture, can be made using the same methods as for stand-alone reconfigurable devices. Usually, this is done by (hand)-mapping important application kernels to the RU and determining the resulting performance by simulation. Determining the *system-level impact* of the RU architecture on the one hand, and the integration of the RU with the CPU and the memory hierarchy on the other hand are challenging tasks. This is because of the complex interactions of the application parts scheduled to run on the CPU and the parts scheduled to run on the RU. A system-wide performance evaluation becomes even more demanding when the dynamic effects of multi-level caching, branch prediction, and out-of-order execution are considered.

Recent work on simulation-based performance evaluation of hybrid CPUs has integrated a purely behavioral RU model into a CPU simulator [6–8]. The RU is characterized by its functionality and an execution time model, both described in a high-level programming language. Although well-suited for fast and functional validation, this approach shows limited accuracy due to the simplifications made in modeling the timing properties of

* Corresponding author. Tel.: +41-1-632-5068; fax: +41-1-632-1210.
E-mail addresses: enzler@ife.ee.ethz.ch (R. Enzler), plessl@tik.ee.ethz.ch (C. Plessl), platzner@tik.ee.ethz.ch (M. Platzner).

0141-9331/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2004.06.004

RU execution, the communication between RU and CPU, the configuration handling, etc.

To gather more accurate system-level performance results, we use a structural, cycle-accurate model of the RU. Instead of building a model for one *specific* configuration of the RU, the RU *itself* is modeled, e.g. computational units and routing network as well as circuitry for reconfiguration and input/output (I/O) interfaces are modeled. The RU model is integrated into a cycle-accurate CPU simulator. The resulting co-simulation evaluation methodology has first been proposed in Enzler et al. [9] and allows the user to assess the value of RU design options and integration alternatives on the system level. Specifically, the framework has recently been used to evaluate virtualized, multi-context devices [10].

The main contributions of this paper compared to previous and related work are the following.

- We present a framework for system-level, cycle-accurate performance evaluation of hybrid reconfigurable processors.
- We discuss a hybrid processor for data-streaming applications which attaches a coarse-grained reconfigurable unit to the coprocessor interface of a CPU core. As with SRAM-based FPGAs, the configuration specifies the functionality of the reconfigurable array.
- We investigate architectural design features that enhance the usability of the reconfigurable unit as a dynamic resource, such as multiple contexts, register replication, and hardware context scheduling.
- We present results of a case study, which investigates the system-level impact of these architectural features.

2. Related work

Different types of hybrid CPUs have been proposed in the literature. One classification looks at the coupling between the reconfigurable unit and the CPU. The tightest coupling is achieved by integrating reconfigurable functional units (RFUs) into the datapath of the CPU [6,7,11,12]. RFUs typically operate on the instruction level. The instructions look much like conventional CPU instructions and have similar execution latencies. Attaching the reconfigurable unit to the coprocessor or the memory interface of the CPU leads to a different class of tightly coupled, hybrid CPUs. In these devices, the field-programmable units execute complete kernels of algorithms with latencies of hundreds of CPU cycles rather than single instructions as in the RFU case.

Another classification focuses on the granularity of the logic blocks used in the reconfigurable array. *Fine-grained* FPGA-like structures attached to a CPU core have been proposed to build application-specific coprocessors [13–15]. These architectures are efficient for bit-oriented or specialized arithmetic operations. For implementing algorithms in digital signal processing, where many arithmetic operations on word-sized data need to be performed, reconfigurable structures based on *coarse-grained* processing blocks have been proposed [16–18].

A crucial design parameter for any dynamically reconfigurable computing system is the reconfiguration time, i.e. the time it takes to reconfigure the functionality. The configuration time imposes limits to the applications that can be successfully mapped to reconfigurable hardware. An approach to overcome these limitations are *multi-context devices*. Instead of holding a single configuration as *single-context devices* do, multi-context devices concurrently hold a set of configurations (the contexts) on the chip. This allows for fast context switching, potentially in a single clock cycle. Fine-grained [19–23], as well as coarse-grained [17,18], multi-context architectures have been proposed.

System-level co-simulation for performance evaluation has been used only for hybrid CPUs that integrate RFUs into their datapath, e.g. for the simulation of the OneChip [6], Chimaera [7], and XiRisc [8,11] architectures. RFU instructions have been integrated with CPU simulators by adding new CPU instructions, which are scheduled to a reconfigurable unit and have a latency given by the implementation of the RFU instruction.

In contrast to the RFU-based architectures, we target reconfigurable units attached to the coprocessor interface of a CPU. The long execution latencies of the reconfigurable unit together with data-dependent processing times and dynamic effects of the CPU pipeline and the caches render functional models for the reconfigurable unit unrealistic. To increase the simulation accuracy, we have chosen a system-wide, cycle-accurate, and execution-based simulation.

3. System-level performance evaluation

To achieve a system-level performance evaluation, we integrate two cycle-accurate simulators into one co-simulation environment. This allows us to use the appropriate simulation tool for each simulation task. The requirements for the CPU simulator are high efficiency, cycle accuracy, and the availability of a robust code-generation framework for compiling benchmarking applications. The requirements for the simulator of the reconfigurable unit are cycle accuracy and the possibility for specification on different abstraction levels, particularly the register-transfer and the structural level.

3.1. Architectural assumptions

Our performance evaluation framework is rather general as it makes only the assumption that a coprocessor interface is used to integrate the RU with a conventional RISC CPU core. Data transfers, configuration loading, and execution

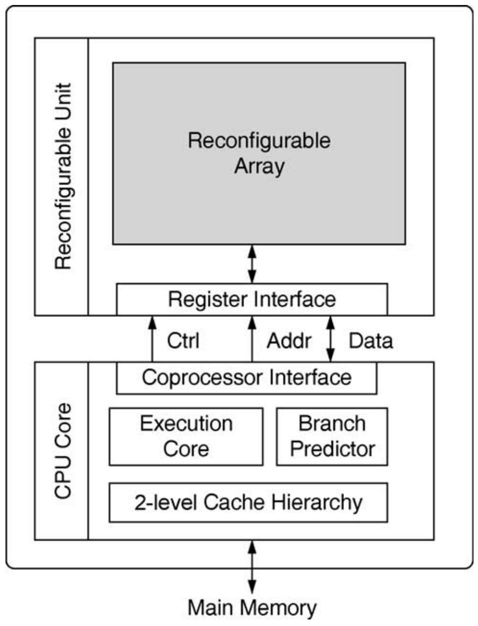


Fig. 1. Hybrid reconfigurable CPU architecture.

control (synchronization of RU and CPU) are performed using the coprocessor interface. Fig. 1 shows the general system architecture.

We rely on the well-established SimpleScalar tool suite [24] for the simulation of the CPU core and the memory architecture. SimpleScalar uses an extended MIPS-like instruction set and compiles applications using a C cross-compiler.

SimpleScalar gathers detailed, cycle-accurate execution statistics by executing the compiled application binary on a parameterized CPU model. The parameters for the superscalar execution core include the number of execution units (fixed-point and floating-point ALUs and multipliers), sizes for the instruction fetch queue (IFQ), the register update unit (RUU), and the load-store queue (LSQ). The simulator supports several kinds of branch predictors, parameterized decode, issue and commit bandwidths, and optional out-of-order execution. The memory architecture can be customized by specifying up to two levels of cache. Overall, this allows for performance evaluation of a broad spectrum of systems, ranging from small, embedded CPUs to high-end, superscalar CPUs. We have added a coprocessor interface and appropriate read and write instructions to the Simple Scalar simulator. The coprocessor interface is modeled as an additional functional unit of the CPU with a dedicated I/O bus.

The reconfigurable unit itself is treated as a black box for system co-simulation. We only require that the RU is modeled using a cycle-accurate VHDL description. The RU simulation is performed on the ModelSim VHDL/Verilog simulator. A conventional VHDL testbench can be used for functional verification of the RU in stand-alone mode. The VHDL model of the RU can be stepwise refined to get a fully synthesizable RU description. The decision for VHDL was driven by its mature development, simulation and synthesis tools. For the future, we consider the SystemC modeling language [25–27] as an interesting alternative, once the tool support for SystemC is on par with VHDL.

3.2. Co-simulation

For co-simulation of the complete hybrid CPU, SimpleScalar must be able to control the RU simulation in VHDL. ModelSim supports the extension of the VHDL simulator through the *foreign language interface* [28]. User-defined shared libraries can be loaded at startup and provide access to the simulation kernel. Making use of this feature, we have implemented an interface that exposes complete control over the RU simulation in ModelSim to the SimpleScalar simulator. Technically, ModelSim and SimpleScalar run as two parallel processes that communicate via commands exchanged using a shared memory area. Whenever SimpleScalar encounters one of the coprocessor instructions, it relays the corresponding command to the ModelSim VHDL simulator. The results are sent back to SimpleScalar. For performance reasons, we allow simulation time on the VHDL simulator to lag behind the CPU simulation time. On every communication event, the simulation times of CPU and RU are re-synchronized.

3.3. Application mapping and toolflow

Mapping an application to the hybrid CPU starts with splitting up the application into parts that run on the CPU core and parts that run on the reconfigurable unit. So far, this partitioning step as well as the specification of the RU configurations is performed manually. The subsequent steps of compiling the application code and generating the configuration bitstreams are automated.

Fig. 2 gives an overview of the co-simulation environment. SimpleScalar requires three input files: the CPU architecture parameters, the compiled application code, and the configuration bitstreams for the RU. The application parts that run on the CPU are implemented in C and compiled with the GCC-based C cross-compiler provided by the SimpleScalar tool suite. The application code needs to take care of downloading the RU configuration bitstreams and of controlling their execution. The new coprocessor instructions, which we have added to SimpleScalar, are accessed using pseudo-assembler instructions. We provide a library that facilitates the access to these pseudo-assembler instructions from a C application. The library contains functions for downloading and switching between contexts, and for transferring data between CPU and RU.

Fig. 3 illustrates how the library functions are implemented: GCC inline assembler macros are used to provide function-like wrappers for the underlying pseudo-assembler instructions. After compiling the C source using these library functions to assembler code, the pseudo-assembler instructions are converted to their binary

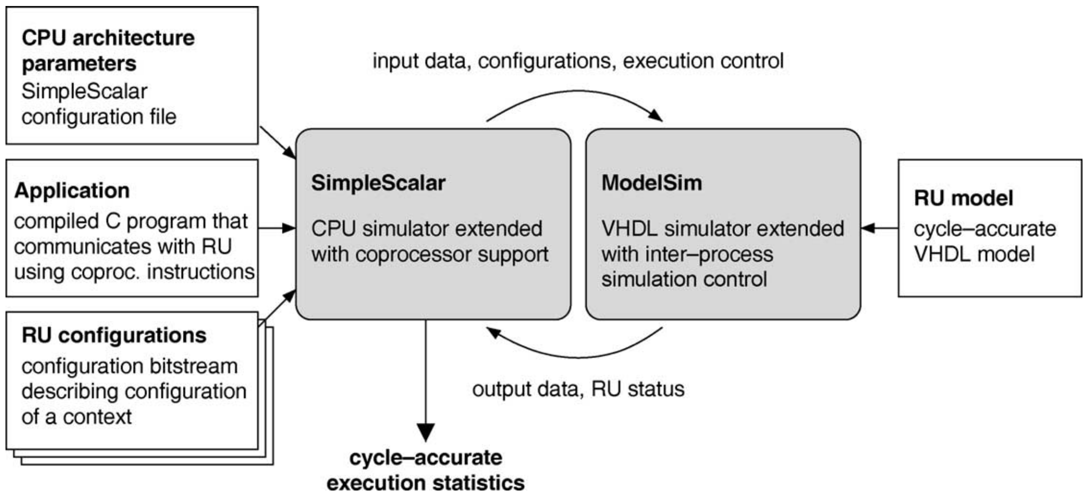

CPU architecture parameters
SimpleScalar configuration file
Application
compiled C program that communicates with RU using coproc. instructions
RU configurations
configuration bitstream describing configuration of a context
input data, configurations, execution control
SimpleScalar
CPU simulator extended with coprocessor support
ModelSim
VHDL simulator extended with inter–process simulation control
RU model
cycle–accurate VHDL model
output data, RU status
cycle–accurate execution statistics

Fig. 2. Co-simulation environment integrating the SimpleScalar and ModelSim simulators.

instruction coding in an intermediate step. This procedure allows introducing new assembler instructions without changing the assembler. However, SimpleScalar must of course be extended to interpret these new instructions correctly.

The functionality of the RU is specified by means of a structural description of the circuit. This structural description is transformed automatically to a configuration bitstream, which the CPU can download to the RU. The RUs VHDL model uses the configuration bitstream to set the functionality of the RU cells and the interconnect.

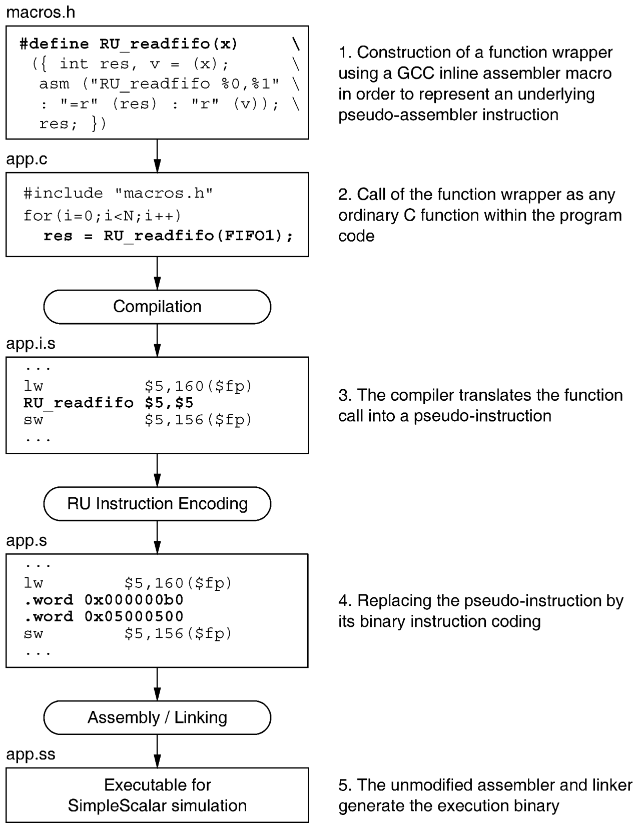
4. Design of the reconfigurable unit

We use the framework described in the last section to investigate and evaluate a hybrid reconfigurable processor for data-streaming applications. In contrast to many approaches studying reconfigurable technology, we aim not at the general-purpose but the embedded computing domain. The embedded domain puts more stringent requirements on computing power, energy consumption, costs, weight, volume, etc. and stresses the trade-offs with respect to these objectives. Consequently, our goal is to employ limited reconfigurable hardware resources in an efficient way.

4.1. General design features

The specific domain of data-streaming applications already points to some design features. First, streaming applications typically make extensive use of integer arithmetic and show a high degree of parallelism [29], which favors coarse-grained reconfigurable arrays. Second, the rather simple memory access patterns ask for FIFO (first-in first-out) data buffers in the reconfigurable unit. Fig. 4 shows the block diagram of the reconfigurable unit. The RU architecture comprises the coprocessor register interface, two FIFO buffers for data transfer, the configuration memory, the context sequencer, and a coarse-grained reconfigurable array. The RU provides a set of coprocessor registers in order to communicate with the CPU core. For example, to access a FIFO, the CPU reads from or writes to the according FIFO coprocessor register.

Two important design features that we investigate are multiple contexts and register replication. Most applications will require more computational resources than the RU provides. Consequently, the application needs to be partitioned into several parts, which are sequentially



macros.h
```
#define RU_readfifo(x)       \
 ({ int res, v = (x);        \
  asm ("RU_readfifo %0,%1"   \
  : "=r" (res) : "r" (v)); \
  res; })
```

app.c
```
#include "macros.h"
for(i=0;i<N;i++)
  res = RU_readfifo(FIFO1);
```

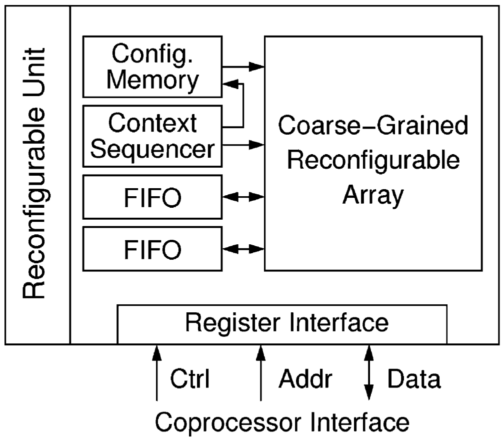
app.i.s
```
...
lw          $5,160($fp)
RU_readfifo $5,$5
sw          $5,156($fp)
...
```

app.s
```
...
lw       $5,160($fp)
.word 0x000000b0
.word 0x05000500
sw       $5,156($fp)
...
```

app.ss

1. Construction of a function wrapper using a GCC inline assembler macro in order to represent an underlying pseudo-assembler instruction
2. Call of the function wrapper as any ordinary C function within the program code
Compilation
3. The compiler translates the function call into a pseudo-instruction
RU Instruction Encoding
4. Replacing the pseudo-instruction by its binary instruction coding
Assembly / Linking
Executable for SimpleScalar simulation
5. The unmodified assembler and linker generate the execution binary

Fig. 3. Toolflow for generating software running on the CPU.



RU coprocessor register	CPU
RU reset	W
FIFO {1,2}	R/W
configuration memory $\{1 \ldots n\}$	W
context select	W
cycle count	R/W
context sequencer start	W
context sequencer status	R
context sequence store $\{1 \ldots s\}$	W

W: write access, R: read access

Fig. 4. Architecture outline of the reconfigurable unit.

executed on the RU. We denote these parts as *logical contexts*. Each context is characterized by its RU configuration. A single-context RU holds exactly one configuration, or *physical context*, on the device. Before a new context can be executed, the CPU must load the corresponding configuration. A multi-context device stores several configurations on-chip. At any time, one of the physical contexts is active and determines the function of the RU.

If the RU cells contain datapath registers, a context carries state. Upon a context switch, the state information, i.e. the content of the datapath registers, must be saved such that it can be restored on the next invocation of the same context. In general, there are three ways to achieve this:

- the content of the datapath register is transferred to a memory, from where it can be read back;
- the state is not explicitly saved and restored but recomputed; or
- the datapath registers are replicated such that each context accesses its own register set, i.e. the state remains in the corresponding registers.

In the following, we concentrate on the latter two alternatives.

Summarized, our architectural model of the RU incorporates several parameterized design features: the datapath width, the depth of the FIFO buffers, the number of contexts the RU can store, and whether the registers are replicated or not.

4.2. Reconfigurable array

The reconfigurable array is a 4×4 array of homogeneous, coarse-grained cells, which are connected by a two-level network (Fig. 5): local interconnects between

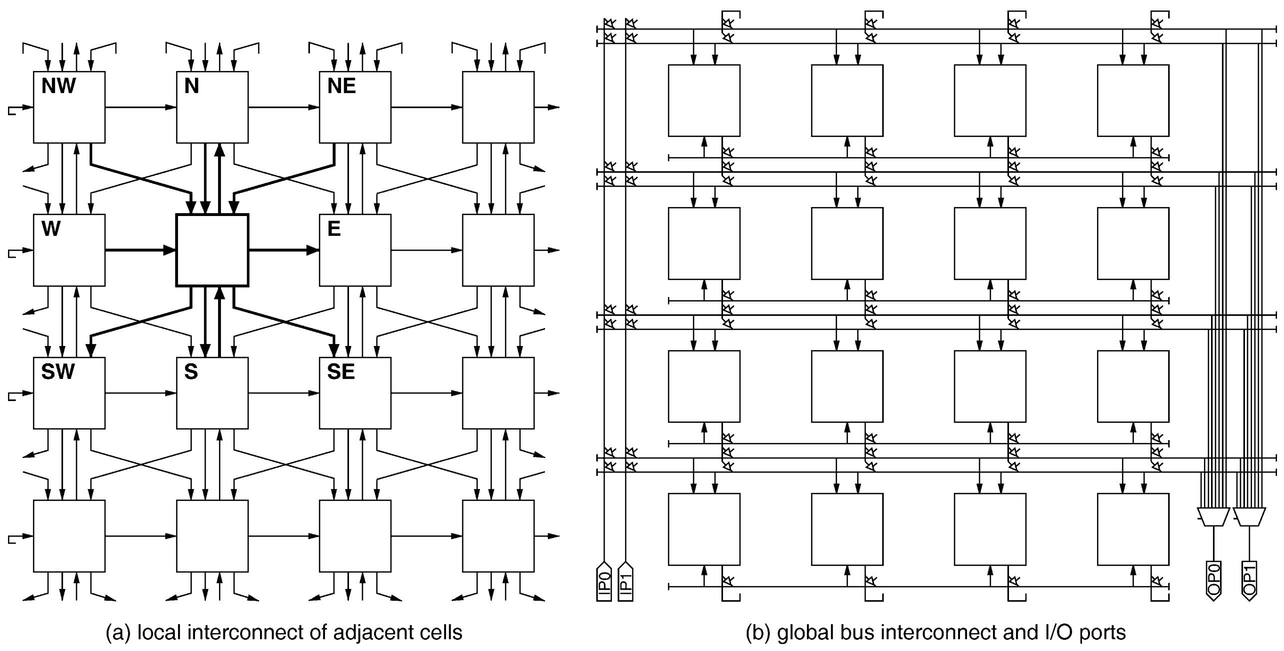

NW
N
NE
W
E
SW
S
SE
IP0
IP1
OP0
OP1
(a) local interconnect of adjacent cells
(b) global bus interconnect and I/O ports

Fig. 5. Two-level interconnect of the reconfigurable array.

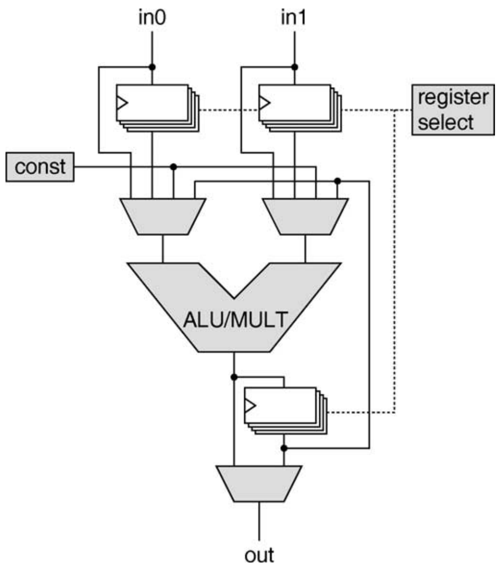


Fig. 6. Datapath of a cell; shaded parts are controlled by the configuration.

certain adjacent cells, and global buses between cell rows. The reconfigurable array has two input and two output ports (IP*x*, OP*x*), which are connected to the two FIFO buffers of the RU. Inside the array, the I/O port interconnects are routed via the global buses.

Fig. 6 outlines the datapath of a cell consisting of a fixed-point arithmetic logic unit (ALU), datapath muliplexers, as well as input and output registers. The registers can be replicated, which allows storing the register state over several context switches. Alternatively, the registers can also be reset upon a context switch. The control signals for the ALU and the multiplexers are part of the configuration. The configuration contains also a constant operator, which can be routed to both ALU inputs. The ALU implements the common arithmetic and logic operations (addition, subtraction, shift, OR, NOR, NOT, etc.) as well as multiplication. If the registers are replicated, the configuration selects further the register plane, which the active context is accessing.

To access the FIFO buffers, the reconfigurable array has to provide the control signal, i.e. read and write enables. To this end, each I/O port has a configurable, fine-grained controller associated, which is depicted in Fig. 7. Each I/O port controller consists of two comparators and a 4-to-1 look-up table (LUT). The inputs are the values of two cycle

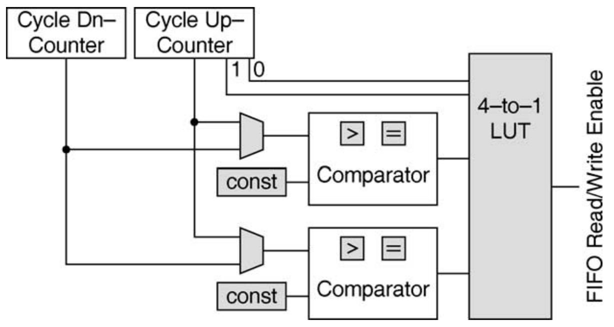


Fig. 7. I/O port controller; shaded parts are controlled by the configuration.

counters, which count the execution cycles computed on the reconfigurable array up, respectively down. The down-counter coincides with the *cycle down* register used for the synchronization of CPU and RU (Section 4.4). Each comparator compares one of the cycle counters to a constant value provided by the configuration. The comparators can be configured to operate in either 'greater than' or 'equal' mode. The outputs of the comparators together with the two least significant bits of the cycle up-counter form the input of the LUT, whose function is specified by the configuration. This mechanism allows generating moderately complex schemes for the FIFO enable signals. Examples are setting the enable signals after a certain amount of computation cycles, or setting them every second or forth cycle.

4.3. Configuration

We assume an SRAM-based configuration technology, as used for fine-grained FPGAs. The configuration is responsible for the functionality of the cells and the I/O port controllers, as well as for the routing of the datapath between the cells, from the input ports to the cells, and from the cells to the output ports. If the datapath registers are replicated, the configuration selects also the active register plane.

Since the configuration incorporates constant values, which may be used in the processing part of the cells, the amount of required configuration bits depends on the parameterized datapath width. Given a datapath width of 16 bit, the configuration data results in 926 bits. The configuration memory holds one or more configurations (the contexts) for the reconfigurable array. The configuration bitstream is written from the CPU to the RU via the configuration interface. The RU supports the download of full and partial configurations for any of the physical contexts.

4.4. Synchronization and context scheduling

The CPU starts the RU execution by writing the number of clock cycles the RU shall perform to the *cycle count* register. In every clock cycle, the *cycle count* register is decremented by one and stops the execution of the RU when reaching zero. Thus, the synchronization mechanism between CPU and RU is similar to the one proposed by Hauser and Wawrzynek [13].

There are two modes how the CPU can control the scheduling of the RU contexts. In the first mode, the CPU selects a context on the RU for execution by writing the context number to the *context selection* register. The context is immediately switched and the CPU can trigger the RU execution by writing the desired number of execution cycles to the *cycle count* register. At any time, the CPU can determine the remaining execution cycles by reading back the value of the *cycle count* register.

In the second mode, the CPU uses the hardware context scheduler (sequencer) incorporated on the RU. The context sequencer consists of the context sequence store and sequence controller. The context sequence store holds a number of sequence instructions that specify the context to be executed, the number of execution cycles, the address of the next sequence instruction to be processed, and the *last context* flag, which determines whether the current context is the last one to be executed. At the beginning of an application, the CPU downloads the sequence instructions into the sequence store. Afterwards, the CPU can trigger the context sequencer by writing to the *context sequencer start* register. The sequencer autonomously executes the sequence instructions and after termination, activates the *sequence status* register.

5. Case study and results

This case study shows how applications can be implemented on an RU with limited hardware resources. The application is partitioned into several logical contexts that are then sequentially processed. In particular, we investigate how multi-context devices can be employed and how the contexts, which carry state information, can be handled. As an example, we present the partitioning and mapping of FIR filters of arbitrary order, which are too large to fit entirely onto the RU. We consider implementations of the same filter on various architectural variants of our hybrid reconfigurable processor.

5.1. FIR filter partitioning and mapping

The answer $Y(z)$ of an FIR filter, given by its transfer function $H(z)$, to an input signal $X(z)$ can be computed as $Y(z)=H(z)\cdot X(z)$. $H(z)$ is a polynomial and, in the FIR filter case, can be factorized into first and second order polynomials, each representing an FIR filter of smaller order. This allows to split up an FIR filter into a cascade of FIR subfilters that are cyclically executed.

Applied to our case study, we implement a 56th-order FIR filter as a cascade of eight subfilters, each of seventh order. Each subfilter stage is mapped to an individual RU context. The filter coefficients are part of the RU configuration. Each FIR subfilter comprises delay registers, which form the state of the RU context. This state must be restored before the same context is executed again. Depending on the capabilities of the reconfigurable array, there are two ways to achieve this:

- If all contexts of the array share the same set of registers, the state must be reconstructed. We achieve this by overlapping subsequent data blocks [30], which results in an execution overhead.
- If the array provides a dedicated set of registers for *each* logical context (full register replication), the state is kept automatically and no overlapping of data blocks is required.

5.2. System setup and experiments

We have set up our hybrid processor architecture to study the following system configurations:

- CPU only (without attached reconfigurable unit),
- CPU with attached single-context RU, and
- CPU with attached 2-, 4-, or 8-context RU.

We assume that CPU and RU operate at the same clock frequency. This assumption is realistic for two reasons: first, we aim at the embedded computing domain, where maximal clock speed is not the sole and major optimization criteria; and second, we use a coarse-grained array allowing for higher clock speeds than fine-grained FPGAs. The CPU model is configured such that it resembles a low-end, embedded CPU (Table 1). We consider reconfigurable arrays with shared datapath registers as well as fully replicated registers.

In each simulation run, 64K samples organized in data blocks are processed. The size of the data blocks depends on the depth of the FIFO buffers available on the RU. We vary the depth of the FIFOs between 64 and 1K words. A data block is written to the RU, processed sequentially by the eight FIR subfilter stages (the eight logical contexts), and then read back. At the beginning of the execution, a control task running on the CPU loads as many contexts as fit onto the RU. If not all logical contexts fit, the contexts are loaded on demand. Each time a filter context is required that is not present on the RU, the CPU control task performs the download by overriding a physical RU context. To switch between logical contexts that are available on the RU, the CPU has two possibilities: either to explicitly initiate a switch by writing to the context select register and then to start the execution of the array, or to make use of the hardware context sequencer. Fig. 8 illustrates the execution flow of various system configurations for a simplified application with three logical contexts:

(a) Only one single physical context is present on the RU, thus the logical contexts must be loaded on demand. After loading a context (Ld C*x*), the CPU starts it

Table 1
CPU model resembling an embedded CPU

CPU parameter	Setup
Execution units	1 int. ALU; 1 int. multiplier; 1 FP ALU; 1 FP multiplier
Caches	32-way 16K L1 I-cache; 32-way 16K L1 D-cache; no L2 cache
Memory interface	32-bit memory bus, 1 memory port
Queue sizes[a]	IFQ: 1; RUU: 4, LSQ: 4
Bandwidths[a]	Decode width: 1; issue width 2; commit width: 2
Instruction issuing	In-order
Branch prediction	Static (always ‘not-taken’)

[a] In number of instructions.

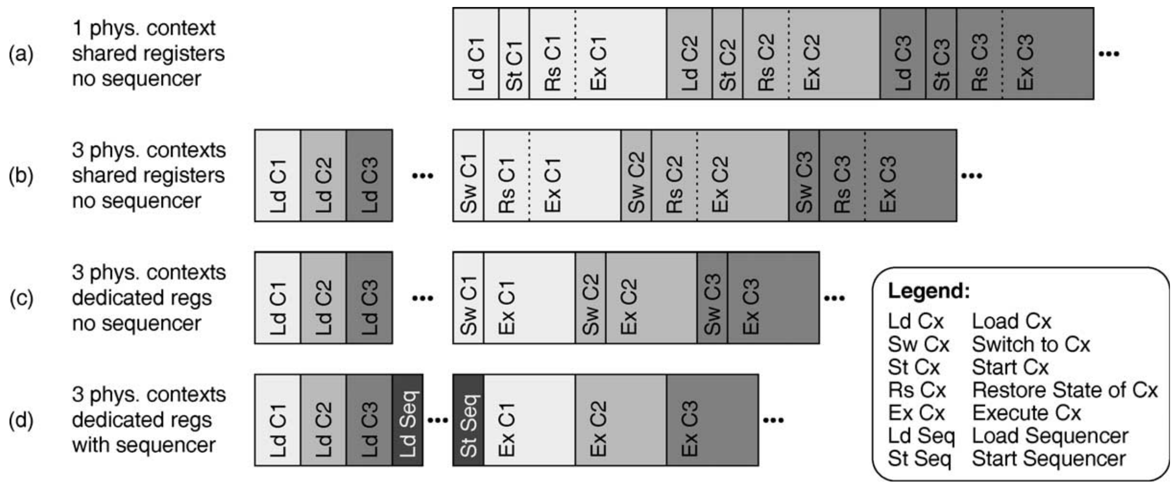

(a) 1 phys. context shared registers no sequencer
Ld C1 St C1 Rs C1 Ex C1 Ld C2 St C2 Rs C2 Ex C2 Ld C3 St C3 Rs C3 Ex C3 ...
(b) 3 phys. contexts shared registers no sequencer
Ld C1 Ld C2 Ld C3 ... Sw C1 Rs C1 Ex C1 Sw C2 Rs C2 Ex C2 Sw C3 Rs C3 Ex C3 ...
(c) 3 phys. contexts dedicated regs no sequencer
Ld C1 Ld C2 Ld C3 ... Sw C1 Ex C1 Sw C2 Ex C2 Sw C3 Ex C3 ...
(d) 3 phys. contexts dedicated regs with sequencer
Ld C1 Ld C2 Ld C3 Ld Seq ... St Seq Ex C1 Ex C2 Ex C3 ...
Legend:
Ld Cx Load Cx
Sw Cx Switch to Cx
St Cx Start Cx
Rs Cx Restore State of Cx
Ex Cx Execute Cx
Ld Seq Load Sequencer
St Seq Start Sequencer

Fig. 8. Execution flow of various system setups for a simplified example with three logical contexts.

(St C*x*). Then, the state of the last activation of this context needs to be restored (Rs C*x*). Finally, the context executes (Ex C*x*). This procedure is cyclically repeated for all logical contexts.

(b) If the RU provides sufficient physical contexts to hold all logical contexts on-chip, the logical contexts are loaded only once at the beginning of the computation. The overhead for activating a new context reduces to switching to the desired context and restoring its state.

(c) If each logical context works on a dedicated set of registers, the context state is kept automatically and does no longer need to be restored.

(d) The on-chip context sequencer reduces the overhead to initiate the start of every sequence (St Seq). The sequencer must be programmed at the beginning of the computation (Ld Seq).

5.3. Results and discussion

Fig. 9 illustrates the results of our experiments as a function of the FIFO buffer depth (shown on the horizontal axis) and the system configuration. All the plotted results are for configurations that make use of the context sequencer.

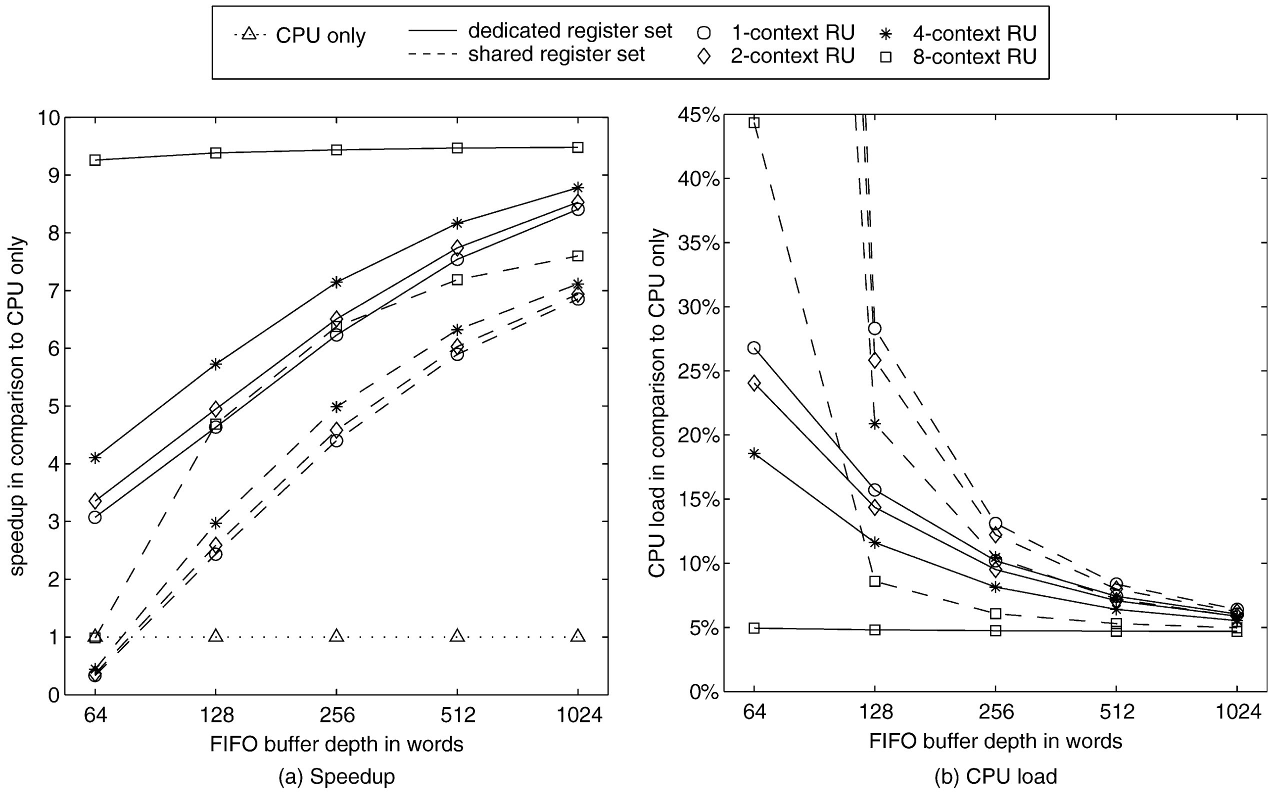


Fig. 9. Performance figures in comparison to the 'CPU only' case.

The execution time of the filter for the 'CPU only' is 110.65 million cycles. Fig. 9(a) shows the speedups relative to this execution time and Fig. 9(b) presents the CPU load normalized to the 'CPU only' case.

Generally, using our reconfigurable unit for an application function shows two benefits. First, we accelerate the function which is measured by the speedup in Fig. 9(a). Second, the CPU is relieved from some operations and can devote the free capacity to other functions. We measure this effect by the relative CPU load in Fig. 9(b). We point out the following observations.

- Using an RU we achieve significant speedups, ranging up to a factor of 9.5 for an 8-context RU with dedicated register sets. The performance deteriorates with decreasing FIFO depth due to the imposed communication overhead.
- Enlarging the FIFOs increases the performance, but at the same time the filter delay. Practical applications could limit these potential gains by imposing delay constraints. For instance, a 2-context RU using a FIFO with 1K words instead of 128 words improves the speedup by a factor of 2.85, while increasing the latency by a factor of 8.
- Full register replication greatly benefits our application as we totally avoid the overlapping of data blocks. For an 8-context RU with a 128-word FIFO, the speedup increases by a factor of 2.0. Additionally, the speedup compared to the 'CPU only' case becomes almost independent of the FIFO depth because no context re-loading is required.
- Employing an RU lowers the CPU load significantly. For a single-context RU with a shared register set the CPU load drops from 100% to 28.3% for a 128-word FIFO and to 6.4% for a 1K-word FIFO. Increasing the number of physical contexts and providing dedicated register sets, the load approaches the asymptotic value of 4.7%, where the CPU task reduces to transferring data and starting the context sequencer.
- As a rather surprising result, we have found that the impact of using the context sequencer is moderate. For an 8-context RU with a 64-word FIFO, the speedup improves by 8.2% and the CPU load drops by 17.1%. However, for fewer physical RU contexts and with increasing FIFO depth the improvement deteriorates. The reason is that our coarse-grained architecture requires only a small amount of context data. With increasing FIFO depth the context switch overhead becomes marginal.

The results from the case study emphasize the importance of the system-level cycle-accurate simulation for architectural evaluation and optimization. As an example, Fig. 9(a) shows that for certain FIFO depths a simple single-context RU with dedicated register sets performs similar or even better than a more involved 8-context RU with a shared register set. The speedup results show that using multiple contexts is valuable. However, for our FIR filter example, register replication is by far more beneficial.

5.4. Simulation speed

A basic characteristic of a simulation environment is the simulation speed. Austin et al. [24] report the simulation speed of SimpleScalar in *simulated instructions per second*. They measured a simulation speed for SimpleScalar of 300K instructions per second. In our case study experiments, we achieve for our embedded CPU model in stand-alone mode, i.e. without attached RU, an average simulation speed of 132K instructions per second on a 900-MHz SunBlade 1000.

For comparison purposes with the hybrid CPU model, we prefer the metric *simulated cycles per second* rather than instructions per second. In the CPU stand-alone mode, we achieve a simulation speed of 226K cycles per second. In the co-simulation case, i.e. when SimpleScalar and ModelSim run concurrently, we achieve an average simulation speed of 4.2K cycles per second. It is important to notice that the simulation speed depends on the load balancing between CPU and RU, which is determined by the application at hand.

6. Conclusion

In this paper, we have presented a C/VHDL co-simulation framework for hybrid reconfigurable processors, which attach a reconfigurable unit to the coprocessor port of a standard CPU core. The cycle-accurate evaluation enables us to study the system-level impact of architectural design features on the performance. We have presented a coarse-grained reconfigurable unit for data-streaming applications. Design features such as multiple contexts, register replication, and context scheduling in hardware have been investigated. In a case study, we have evaluated various system configurations. The results illustrate the importance of the system-level, cycle-accurate performance evaluation. Without such a methodology, the trade-offs involved in reconfigurable processor design could not be satisfyingly analyzed.

Since we target the embedded domain, our goal is to employ limited reconfigurable hardware resources in an efficient way, rather than using devices of arbitrary large size. Considering large FIR filters as example, our case study illustrates how applications can be virtualized by partitioning them into logical contexts that are sequentially processed. We particularly study the usage of multi-context reconfigurable arrays and ways how the contexts, which carry state information, can be handled.

Future work includes the analysis of applications that require more complex context sequences (control flow), the integration of a dedicated RU memory port, and the investigation of context prediction and prefetching techniques. We also aim at the development of an area model for the RU in order to quantify the hardware overhead introduced by certain architectural design features, e.g. the multi-context support.

Acknowledgements

This work is supported by the Swiss Federal Institute of Technology (ETH) Zurich under the ZIPPY project and the Wearable Computing Polyproject.

References

[1] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, J. Rabaey, Evaluation of a low-power reconfigurable DSP architecture, in: Proceedings of the Fifth Reconfigurable Architectures Workshop (RAW), vol. 1388 of LNCS, Springer, Berlin, 1998 pp. 55–60.

[2] O. Mencer, M. Morf, M.J. Flynn, Hardware software tri-design of encryption for mobile communication units, in: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 5 1998 pp. 3045–3048.

[3] K. Compton, S. Hauck, Reconfigurable computing: a survey of systems and software, ACM Computing Surveys 34 (2) (2002) 171–210.

[4] K. Bondalapati, V.K. Prasanna, Reconfigurable computing systems, Proceedings of the IEEE 90 (7) (2002) 1201–1217.

[5] F. Barat, R. Lauwereins, Reconfigurable instruction set processors: a survey, in: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP), 2000 pp. 168–173.

[6] J.E. Carrillo Esparza, P. Chow, The effect of reconfigurable units in superscalar processors, in: Proceedings of the Ninth ACM International Symposium on Field-Programmable Gate Arrays (FPGA), 2001 pp. 141–150.

[7] Z.A. Ye, A. Moshovos, S. Hauck, P. Banerjee, CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit, in: Proceedings of the 27th International Symposium on Computer Architecture (ISCA), 2000 pp. 225–235.

[8] A. La Rosa, L. Lavagno, C. Passerone, A software development tool chain for a reconfigurable processor, in: Proceedings of the Fourth International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2001 pp. 93–98.

[9] R. Enzler, C. Plessl, M. Platzner, Co-simulation of a hybrid multi-context architecture, in: Proceedings of the Third International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), CSREA Press, Las Vegas, NV, 2003 pp. 174–180.

[10] R. Enzler, C. Plessl, M. Platzner, Virtualizing hardware with multi-context reconfigurable arrays, in: Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL), vol. 2778 of LNCS, Springer, Berlin, 2003 pp. 151–160.

[11] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri, A VLIW processor with reconfigurable instruction set for embedded applications, IEEE Journal of Solid-State Circuits 38 (11) (2003) 1876–1886.

[12] R. Razdan, M.D. Smith, A high-performance microarchitecture with hardware-programmable functional units, in: Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27), 1994 pp. 172–180.

[13] J.R. Hauser, J. Wawrzynek, Garp: a MIPS processor with a reconfigurable coprocessor, in: Proceedings of the Fifth IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1997 pp. 12–21.

[14] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, M. Gokhale, The NAPA adaptive processing architecture, in: Proceedings of the Sixth IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1998 pp. 28–37.

[15] Xilinx Inc., Virtex-II Pro Platform FPGA Handbook (January 2002).

[16] T. Miyamori, K. Olukotun, REMARC: reconfigurable multimedia array coprocessor, IEICE Transactions on Information and Systems E82-D (2) (1999) 389–397.

[17] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, E.M. Chaves Filho, MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications, IEEE Transactions on Computers 49 (5) (2000) 465–481.

[18] B. Salefski, L. Caglar, Re-configurable computing in wireless, in: Proceedings of the 38th Design Automation Conference (DAC), 2001 pp. 178–183.

[19] A. DeHon, DPGA utilization and application, in: Proceedings of the Fourth ACM International Symposium on Field-Programmable Gate Arrays (FPGA), 1996 pp. 115–121.

[20] S. Trimberger, D. Carberry, A. Johnson, J. Wong, A time-multiplexed FPGA, in: Proceedings of the Fifth IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1997 pp. 22–28.

[21] T. Fujii, K.-i. Furuta, M. Motomura, M. Nomura, M. Mizuno, K.-i. Anjo, K. Wakabayashi, Y. Hirota, Y.-e. Nakazawa, H. Itoh, M. Yamashina, A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture, in: 46th IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers, 1999 pp. 364–365.

[22] J. Faura, J.N. Moreno, M.A. Aguirre, P. van Duong, J.M. Insenser, Multicontext dynamic reconfigurable and real-time probing on a novel mixed signal programmable device with on-chip microprocessor, in: Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications (FPL), vol. 1304 of LNCS, Springer, Berlin, 1997 pp. 1–10.

[23] S.M. Scalera, J.R. Vázquez, The design and implementation of a context switching FPGA, in: Proceedings of the Sixth IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1998 pp. 78–85.

[24] T. Austin, E. Larson, D. Ernst, SimpleScalar: an infrastructure for computer system modeling, IEEE Computer 35 (2) (2002) 59–67.

[25] P.R. Panda, SystemC: a modeling platform supporting multiple design abstractions, in: Proceedings of the 14th International Symposium on Systems Synthesis (ISSS), 2001 pp. 75–80.

[26] T. Grötker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Kluwer Academic Publishers, Dordrecht, 2002.

[27] Open SystemC Initiative (OSCI), SystemC 2.0 User's Guide, available at http://www.systemc.org/ (2002).

[28] Model Technology Inc., ModelSim Foreign Language Interface, Version 5.5f (August 2001).

[29] R. Enzler, M. Platzner, C. Plessl, L. Thiele, G. Tröster, Reconfigurable processors for handhelds and wearables: application analysis, in: Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications III, vol. 4525 of Proceedings of SPIE, 2001 pp. 135–146.

[30] A.V. Oppenheim, R.W. Schafer, Discrete-Time Signal Processing, second ed, Prentice-Hall, Englewood Cliffs, NJ, 1999.

Rolf Enzler received the Dipl.-Ing. (MSc) degree in electrical engineering from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, in 1997. In 1997, he joined the Electronics Laboratory at ETH Zurich, where he is currently pursuing his PhD. His research interests include reconfigurable computing, embedded systems, and application-specific computing architectures.

Christian Plessl received the Dipl.-Ing. (MSc) degree in electrical engineering from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, in 2001. Since then, he is a research assistant and PhD student at the Computer Engineering and Networks Laboratory at ETH Zurich. His research interests include reconfigurable computing for embedded systems and custom computing machines.

Marco Platzner received the Dipl.-Ing. and the PhD degrees in telematics from Graz University of Technology, Graz, Austria, in 1991 and 1996, respectively. From 1991 to 1996, he was with the Institute for Technical Informatics, Graz University of Technology, where he worked on embedded multiprocessors for simulation and digital signal processing applications. From 1996 to 1998, he held a post-doctoral researcher position at GMD—German National Research Center for Information Technology, St Augustin, Germany, where he developed robot vision systems for autonomously moving vehicles. From 1997 to 1998, he was a visiting scholar at the Computer Systems Laboratory (CSL), Stanford University, Stanford, CA. In 1998, Marco Platzner joined the Computer Engineering and Networks Laboratory at the Swiss Federal Institute of Technology (ETH) Zurich, where he is leading several projects in the area of reconfigurable systems. His current research interests include reconfigurable computing, hardware–software codesign, and embedded systems.