Co-simulation of a Hybrid Multi-Context Architecture

Rolf Enzler, Christian Plessl, and Marco Platzner
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
E-mail: enzler@ife.ee.ethz.ch

***Abstract*— Reconfigurable computing architectures aim to dynamically adapt their hardware to the application at hand. As research shows, the time it takes to reconfigure the hardware forms an overhead that can significantly impair the benefits of hardware customization. Multi-context devices are one promising approach to overcome the limitations posed by long reconfiguration times. In contrast to more traditional reconfigurable architectures, multi-context devices hold several configurations on-chip. On demand, these devices quickly switch to another context. In this paper we present a co-simulation environment to investigate design trade-offs for hybrid multi-context architectures. Our architectural model comprises a reconfigurable unit closely coupled to a CPU core. As a case study, we discuss the implementation of an FIR filter partitioned into several contexts. We outline the mapping process and present simulation results for single- and multi-context reconfigurable units coupled with embedded and superscalar CPUs.**

***Index Terms*— reconfigurable computing, multi-context, field-programmable gate array (FPGA), co-simulation, VHDL.**

I. Introduction

Reconfigurable computing dynamically adapts the computational hardware of a processing element to the application at hand. Through hardware customization we can achieve a performance close to that of application-specific hardware architectures. The dynamics of the customization process results in a flexibility close to that of software programmable processors.

The time it takes to reconfigure a device is a crucial parameter for any dynamically reconfigurable computing system. The configuration time poses limits to the applications that can be successfully mapped to reconfigurable hardware. Commodity reconfigurable devices, especially field-programmable gate arrays (FPGAs), have relatively long configuration times in the range of dozens of milliseconds. Consequently, rather long-running application functions are mapped to such devices. For smaller functions the configuration overhead can be significant, nullifying any performance gains over software. The trend towards ever larger devices aggravates the problem further, because the configuration time is proportional to the amount of configuration data which in turn grows with the device size.

For single-context devices, i. e. devices that store one set of configuration data on the chip, several improvements of the reconfiguration mechanism have been proposed, including partial reconfiguration [1], configuration compression [2], and configuration prefetching [3]. Partial reconfiguration allows to reconfigure only parts of the device, which results in smaller configuration streams. Usually, existent configurations on the device stay operational when a partial configuration is downloaded to another part of the device. Configuration compression intends to minimize the amount of configuration data by employing lossless data compression techniques. The compressed configuration stream is decompressed by the device on-the-fly during the configuration process. Configuration prefetching targets devices where an attached processor controls the download of the configuration data. The intention is to write the next required configuration as early as possible to the reconfigurable device, such that the configuration is ready when invoked by the program running on the processor.

This work is supported by ETH Zurich under the ZIPPY project and the Wearable Computing Polyproject.

Multi-context devices are another approach to overcome the limitations posed by long configuration times. Instead of holding a single configuration, multi-context devices concurrently hold a set of configurations on the chip. At any given time there is exactly one configuration in use, the so-called active context. Holding several contexts on-chip has several advantages. Switching to another context is very fast, potentially a single clock cycle. Further, the latency of writing a context to the device can often be hidden, partially or entirely, by writing a context while another context is active. Further optimizations, e. g. partial configuration, configuration compression or prefetching, can also be applied in multi-context devices.

Fast reconfiguration enables novel models of computation that are not achievable by traditional configurable devices, such as emulation of arbitrary large reconfigurable devices and time sharing [4]. Reconfigurable devices of arbitrary capacity can be emulated by switching between several contexts, where each context implements a part of a single large design. In the time sharing case, on the other hand, several applications compete for the reconfigurable hardware resources. Each application receives a certain share of the FPGA's execution time and the applications are sequentially swapped in and out.

Although the multi-context concept promises to be a valuable extension for configurable devices, a number of pressing research issues stay open. The major question is whether the shorter reconfiguration time pays off the introduced hardware overhead. The trade-offs between performance gains and increased multi-context resources have to be studied and the operation modes identified, which benefit most from the multi-context facilities. Another issue is the system integration, including the coupling of the multi-context device to a controller, the efficient transfer of data, and the sequencing of context switches. A further aspect are tools supporting multi-context devices. This comprises the automatic partitioning of applications as well as scheduling schemes for contexts.

This paper concentrates on the system integration issue and, by means of a case study, illustrates the computation model of emulating an arbitrary large device. We present a hybrid architecture that closely couples a multi-context reconfigurable unit (RU) to a processor core. In contrast to most of the published work, we gather realistic experimental data by employing a *cycle-accurate co-simulation* environment. As a case study, we discuss the partitioning and mapping of an FIR filter onto the reconfigurable unit. We present simulation results for single- and multi-context reconfigurable units coupled with embedded and superscalar CPUs.

Section II surveys related work. Section III outlines the system architecture and describes the RU model. Section IV explains the C/VHDL co-simulation environment. Section V presents the FIR filter example. Finally, Section VI concludes the paper and outlines future work.

II. Related Work

WASMII was one of the first architectures that proposed to extend an FPGA to hold several context planes and included a virtual hardware concept [5]. Based on WASMII, a prototype chip called Dynamically Reconfigurable Logic Engine (DRLE) was developed [6] and the virtual hardware concept was implemented [7]. DeHon outlined the Dynamically Programmable Gate Array (DPGA) architecture and proposed to couple this multi-context device with a microprocessor [8]. A DPGA prototype chip was presented in [9]. Trimberger *et al.* outlined a multi-context extension of the Xilinx XC4000E FPGA and discussed appropriate operation modes [4]. FIPSOC (Field-Programmable System On Chip) is a hybrid architecture with two contexts, which are mapped directly into the microprocessor's address space [10]. The Context Switching Reconfigurable Computing (CSRC) device is a hierarchically structured FPGA architecture [11]. Several applications were mapped onto a CSRC device in [12]. All these devices are fine-grained architectures. In contrast to that, we model a hybrid coarse-grained architecture. Chameleon Systems' CS2000 is a coarse-grained device, which couples a processor with a 32-bit reconfigurable fabric [13]. The fabric holds a background plane, which can be loaded while the active plane is in use. The CS2000 differs from our work in the construction of the reconfigurable unit and the coupling of reconfigurable unit and processor.

Co-simulation of hybrid multi-context devices has mainly been done by including a purely functional model of the reconfigurable unit into a CPU simulator. In [14], OneChip is described as a hybrid architecture that integrates a reconfigurable functional unit into the pipeline of a superscalar RISC processor. Similar to our work, the architecture is simulated using an extension to the SimpleScalar CPU simulator. In contrast to our approach, the array itself is not modeled explicitly. Instead, the functionality and execution latency of a particular array configuration is specified. We have chosen a cycle-accurate co-simulation approach that explicitly models the array architecture. Its functionality is determined by the configuration bitstream.

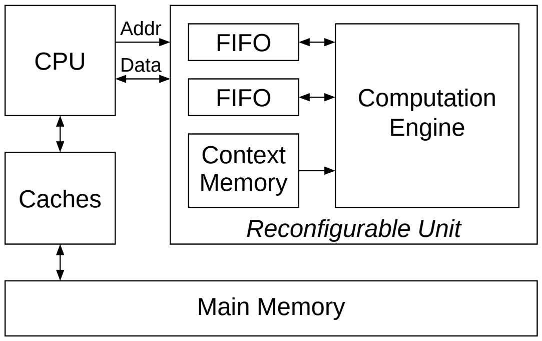


Fig. 1. System model outline.

III. Hybrid Multi-Context Architecture

Fig. 1 outlines the basic system model, which comprises a CPU core, instruction and data caches, and the reconfigurable unit (RU). The RU is attached to the CPU via a dedicated coprocessor interface. For system co-simulation, we use a cycle-accurate CPU model written in C and an RU model specified in VHDL. This section describes the architectural issues, the next section the co-simulation issues.

A. CPU Architecture Model

We employ the SimpleScalar processor simulator which is based on a 32-bit RISC processor architecture and a MIPS-like instruction set [15]. The CPU core as well as the two-level cache hierarchy are widely configurable. The major parameters of the CPU core are

- the number of computation units (integer and floating-point ALUs and multipliers),
- the sizes of the instruction fetch queue (IFQ), the register update unit (RUU), and the load/store queue (LSQ),
- decode, issue, and commit bandwidths,
- in-order or out-of-order execution, and
- the branch prediction mode.

The main parameters of the on-chip data and instruction caches are cache size, number of sets, associativity, and replacement strategy. Due to these configuration options, the CPU model can be parameterized such that it resembles a broad range of architectures, from small low-end to powerful high-end CPUs.

B. RU Architecture Model

The RU model consists basically of two FIFO buffers, the context memory, and the computation engine (Fig. 1). The model incorporates a set of parameterizable characteristics: the datapath width (up to 32 bits), the depth of the FIFO buffers, and the number of configurations the context memory can hold. Consequently, the model can be parameterized to represent different RU variants.

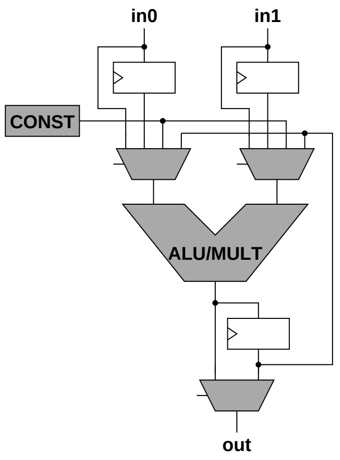


Fig. 2. Processing part of a cell. The shaded components are controlled by the configuration.

C. Coupling of CPU and Reconfigurable Unit

To attach the RU to the CPU, we have extended SimpleScalar's CPU model with a coprocessor interface. To this end, we have added two new instructions to the SimpleScalar instruction set. The *write to coprocessor register* instruction takes a value and address from the CPU register file and writes the value to the coprocessor register with the given address. The *read value from coprocessor register* instruction works accordingly, reading a value from a specified coprocessor register. Data to and from the RU is transfered via the FIFO buffers. Both FIFOs are readable and writable by the CPU as well as the RU.

The synchronization mechanism between CPU and RU is similar to the one proposed in the Garp processor [16]. The execution of the RU is started 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. As our CPU model does not support interrupts so far, the synchronization of CPU and RU execution is done by polling the cycle count register.

D. Multi-Context Support

The context memory holds a set of entire configurations for the computation engine. The CPU writes the configuration data via coprocessor interface in 32-bit chunks to the RU. This mechanism allows writing an entire configuration as well as a partial reconfiguration.

Before the CPU initiates a computation on the RU, it writes the number of the context to be activated to the *context selection register*. The context is immediately switched and the CPU can start the RU computation by writing the *cycle count register*.

E. Computation Engine

The computation engine is a 4×4 array of homogeneous, coarse-grained cells, which are connected by a 2-level network. The cells consist of a processing and a routing part.

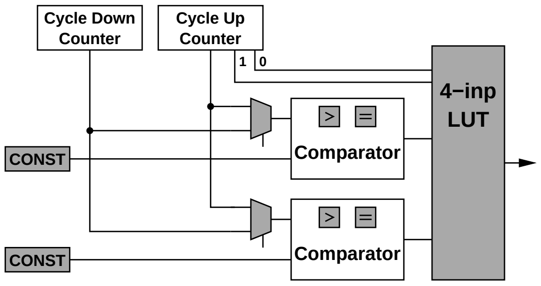


Fig. 4. Controller of a computation engine I/O port. The shaded components are controlled by the configuration.

Fig. 2 outlines the processing part of a cell. It consists of a fixed-point arithmetic logic unit (ALU), datapath multiplexers, as well as input and output registers. The control signals for the ALU and the multiplexers are part of the configuration. The configuration contains furthermore a constant operator, which can be routed to both ALU inputs. Both inputs and the output can be registered. The ALU implements the common arithmetic and logic operations (addition, subtraction, shift, OR, NOR, NOT, etc.) as well as multiplication.

The cell interconnect has two routing levels (Fig. 3): direct connections between certain adjacent cells and horizontal buses between cell rows. The interconnect is cyclically continued at the array borders. Each cell can route the outputs of five of its neighbors via the direct connections to its inputs. The inputs can also be connected to the three horizontal buses located between each cell row. Two of the horizontal buses can be driven by the cells from the row above. The third horizontal bus can be driven by the cells in the same row, which allows for feeding back the outputs of the same row.

The computation engine features two input and two output ports, which are connected to the two FIFOs of the RU. Fig. 3(b) shows the routing options inside the computation engine. The input and output ports can be routed via two of the three horizontal buses between each cell row.

To access the FIFOs, the computation engine has to provide the control signals, i. e. read and write enables. To this end, each I/O port has a configurable fine-grained controller associated (Fig. 4). Each I/O port controller consists of two comparators and a 4-to-1 look-up table (LUT). The inputs are the cycle down counter discussed in Section III-C, as well as a cycle up counter, counting contrariwise. 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 are the inputs of the LUT, whose functionality is part of the configuration. This concept allows a simple generation of moderately complex schemes for the FIFO enable signals. Examples are setting the enable signals after a certain amount of computation cycles, or setting them a certain amount of cycles before the end of a computation.

Summarized, the configuration of the computation engine is responsible for the functionality of the cells and the I/O port

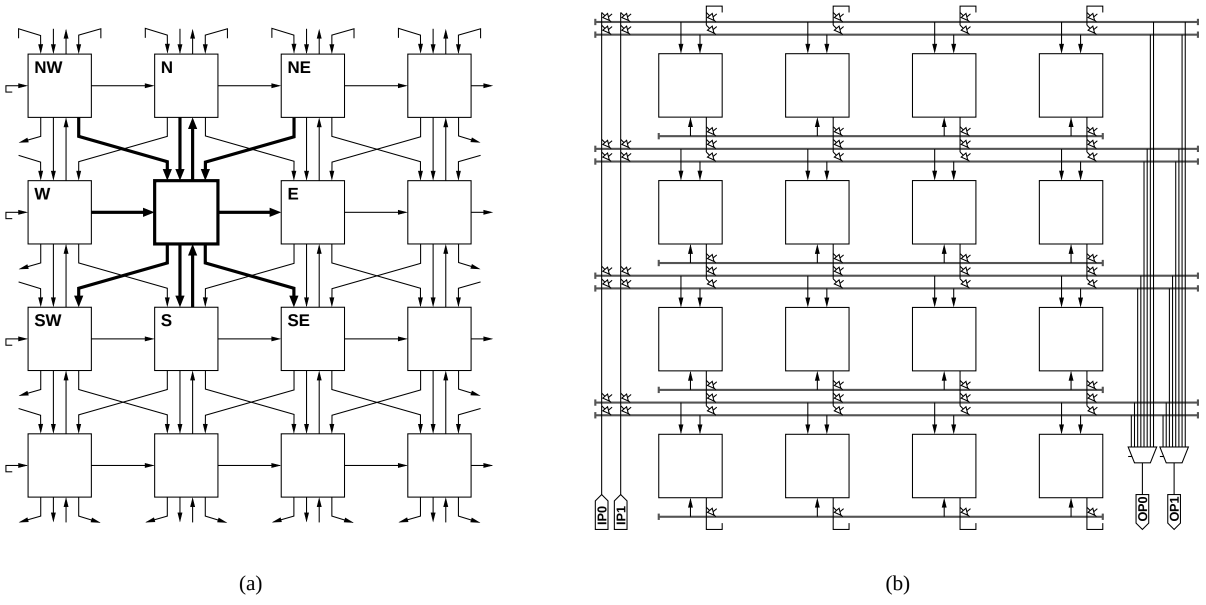


Fig. 3. 2-level interconnect scheme of the computation engine: (a) direct connections (highlighted the connections of one cell), and (b) horizontal buses and I/O ports (IPx, OPx).

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. 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 datapath width. Given a datapath width of 16 bit, the configuration data results in 918 bits.

IV. Co-Simulation Environment

In order to simulate CPU and RU, we employ two different simulators integrated into a co-simulation environment. This concept allows us to use the appropriate tool for every simulation task. For the CPU simulation, tool support for generating executables that run on the simulator is essential. For that reason our co-simulation environment is based on the well-established SimpleScalar processor simulator [15].

A. CPU Simulation Model

SimpleScalar is an extensible, parameterizable CPU simulation tool suite that simulates a superscalar, out-of-order issue CPU with a MIPS-like instruction set called PISA. SimpleScalar provides cycle-accurate simulation, supports multi-level cache architectures, and provides detailed simulation statistics. Details on the simulation statistics and measurement results of various applications can be found in [17].

The simulator is available in C source code and allows for modification for academic purposes. SimpleScalar can be easily extended with new instructions and even functional units. The RU attached to the CPU is modeled as an additional SimpleScalar functional unit. This adds a dedicated I/O bus with its own I/O address space to the CPU model and enables concurrent accesses on the memory and the coprocessor bus.

B. RU Simulation Model

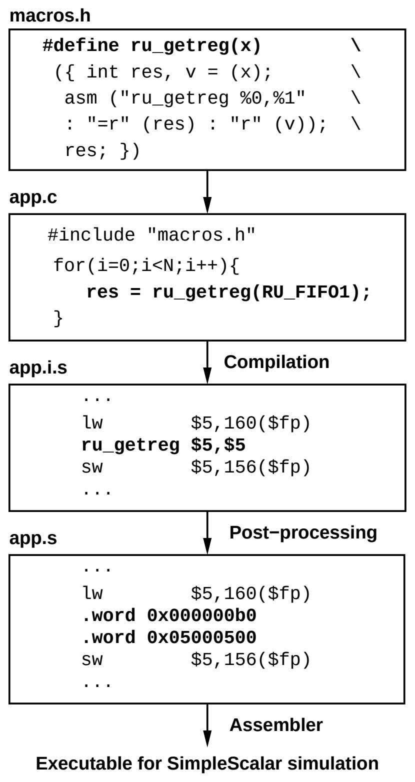
The RU is modeled in VHDL. Using a VHDL simulator instead of writing an RU simulator from scratch has several advantages. Building on the discrete event simulation kernel provided by the VHDL simulator simplifies the implementation of the model. Furthermore, VHDL allows to model the system on different levels of abstraction. Parts of the RU model may be implemented in non-synthesizable behavioral VHDL, where parts of the model are specified on register transfer (RTL) level. If a prototype of the RU shall be implemented, the RU model can be stepwise converted to the synthesizable VHDL subset. Sophisticated VHDL simulators ease the development of the RU due to the availability of waveform viewers and VHDL debuggers.

C. Co-simulation of CPU and RU

We use ModelSim 5.6 for VHDL simulation. ModelSim is a powerful VHDL/Verilog simulator that can be extended via the *foreign language interface*. Essentially this allows access to the simulation kernel by building a shared library, which is loaded by the simulator at startup.

In order to simulate the RU in stand-alone mode, ModelSim can be used with a conventional VHDL testbench, which drives the inputs of the RU and interprets the output signals. We use this method for the verification of the RU model.

For co-simulating CPU and RU, control of the RU simulation from within the SimpleScalar simulator is needed. To this end we have implemented an interface using the ModelSim foreign language interface that exposes complete control of the simulation to SimpleScalar. The SimpleScalar and ModelSim processes communicate by the means of messages, which are exchanged using a shared memory area.



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

```
#include "macros.h"
for(i=0;i<N;i++){
   res = ru_getreg(RU_FIFO1);
}
```

```
...
lw        $5,160($fp)
ru_getreg $5,$5
sw        $5,156($fp)
...
```

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

Fig. 5. Software Toolfbw.

The access to the shared memory area is synchronized with a simple handshake protocol implemented with semaphores. Whenever SimpleScalar encounters one of the coprocessor instructions it relays the corresponding command to the VHDL simulator, where it is processed. The results are then sent back to SimpleScalar. For performance reasons we allow simulation time on the VHDL simulator to lag behind the CPU simulation time. At every communication the simulation time of CPU and RU is re-synchronized.

D. Software Toolflow for Applications

The SimpleScalar tool suite provides versions of the GNU C compiler and the GNU binutils that support the PISA instruction set used by SimpleScalar. Although SimpleScalar allows for easy modification of the instruction set that is simulated, the corresponding tools (compiler, assembler) cannot be generated automatically. Thus, we have chosen not to modify these tools directly, but to add some intermediate steps to the code generation in order to support the new coprocessor instructions we have added to SimpleScalar. The programmer can use the new coprocessor instructions by means of pseudo assembler instructions. Before assembling, these pseudo assembler instructions are translated to their binary instruction encoding. Subsequently, the resulting assembler files can be assembled with the ordinary SimpleScalar assembler. To simplify the usage of these coprocessor instructions in applications written in C, function wrappers have been implemented using template-based inline assembler macros provided by GNU C.

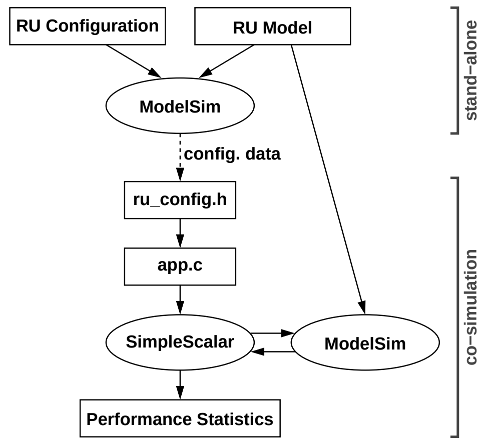


Fig. 6. Hardware Toolfbw.

Fig. 5 outlines the software toolflow. It illustrates how the *ru_getreg* pseudo-instruction, which was added to the SimpleScalar instruction set, can be used within a C application. A function wrapper using GNU inline assembler is defined in *macros.h*. The function can be called within *app.c* like any ordinary C function, but is translated by the compiler to a *ru_getreg* pseudo-instruction. The compiler takes care of inserting the correct register names to the inline assembler template. Before passing this intermediate file to the assembler, the pseudo-instruction is replaced by its binary instruction coding. The resulting assembler file can then be processed by the unmodified assembler and linker.

E. Hardware Toolflow for Applications

Fig. 6 shows the hardware toolflow, i. e. the part of the application implementation that concerns the reconfiguration unit. The RU configurations are generated manually so far. A single RU configuration is specified by means of a VHDL record, which represents a structured view of a configuration.

For the stand-alone simulation of the RU the VHDL records are used directly to specify the RU configuration. For the co-simulation of CPU and RU a configuration bitstream that can be downloaded from the CPU to the RU is automatically generated. The configuration bitstream is dumped into the *ru_config.h* file, which is included by the application. This concept ensures the consistency of the configuration data throughout the whole toolflow.

V. Example: FIR Filtering

This section illustrates how fast reconfiguration enables the emulation of arbitrary large reconfigurable devices. As an example, we show how FIR filters can be partitioned and mapped onto RU contexts in order to implement filters of arbitrary order on a device with limited hardware resources. Based on this background, we study implementations of the same filter on different variants of our architecture. With an embedded CPU model as a starting point, we investigate the benefit of employing additional hardware resources by attaching a reconfigurable unit or by using a more sophisticated CPU.

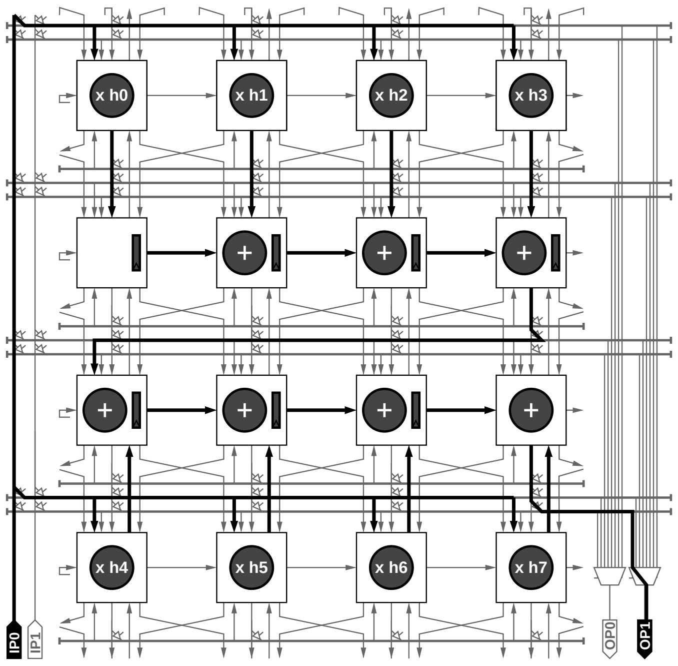


Fig. 7. One 8-tap FIR filter stage mapped onto the computation engine.

A. 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 in z^{-1}, given by

$$H(z) = h_0 + h_1 z^{-1} + \ldots + h_m z^{-m} = \sum_{i=0}^{m} h_i \cdot z^{-i},$$

and can be factorized into smaller polynomials, each of which represents an FIR filter of smaller order. Since all coefficients h_i of $H(z)$ are real, $H(z)$ can be factorized into first and second order polynomials $H_i(z)$, $H(z) = H_1(z) \cdot H_2(z) \cdot \ldots \cdot H_l(z)$. This naturally allows to split up an FIR filter into a cascade of FIR filters (sections), where each section implements a part of the factorized transfer function.

In our example we implement a 56-order FIR filter as a cascade of eight 8-tap filter stages (each of order 7). Each stage is implemented in transposed direct form. Fig. 7 shows a simplified schematic of the mapping of one of these stages onto the computation engine. The coefficients of the FIR filter are part of the configuration.

B. Experimental Setup

We employ two different CPU models: an *embedded CPU*, which is modeled after an Intel StrongARM processor, and a more powerful *superscalar CPU*, which corresponds to SimpleScalar's default configuration. Table I summarizes the main architectural parameters of the two CPU models. We assume that both CPUs feature the same technology figures.

With respect to the coprocessor, we study two cases: a *single-context RU* and a *multi-context RU* holding eight contexts on-chip, both incorporating FIFO buffers of depth 256.

C. Results

In total we process 64K samples organized in data blocks of 256 samples. The data blocks are processed sequentially. In order to avoid transient filter effects at the borders of the data blocks, we overlap the blocks.

In the coprocessor cases, a data block is written to the RU, processed sequentially by the eight FIR filter stages (the eight contexts), and read back. The RU registers are reset upon a context switch to prevent that the register contents of the previous context affect the current context. With the multi-context RU, the CPU downloads the eight contexts at the beginning and then switches between them. With the single-context RU, the CPU has to download the required context after each filter stage.

Table II shows the performance results of the six studied setups. SimpleScalar reports the execution time in terms of cycles. In the case of the FIR filters, an often used performance metric is $cycles/(tap * sample)$. In order to provide a fair comparison, we have normalized all results to the filter implementation running on the CPUs without attached coprocessor. Since in this case the 56-order FIR filter has not to be partitioned, it can be implemented with 57 taps.

The results show that employing a more powerful CPU or attaching a reconfigurable coprocessor results in significant speedups. The following observations can be made:

- the superscalar CPU yields a speedup of about 3 over the embedded CPU,
- attaching a coprocessor results in significant speedups for both embedded and superscalar CPU, and
- the multi-context reconfigurable unit shows additional potential compared to the single-context unit.

An important insight is that the embedded CPU with attached coprocessor is competitive to the superscalar CPU. This shows that reconfigurable hardware is a valuable option to achieve performance gains.

It has to be pointed out that in our experiment the benefit of the multi-context over the single-context unit significantly depends on the ratio between configuration data and the number of samples in a data block. We use a rather small, coarse-grained RU with modest amount of configuration data, which limits the achievable speedup.

VI. Summary and Future Work

In this paper, we have presented a C/VHDL co-simulation framework for a hybrid multi-context architecture. We have discussed a hardware and software toolflow that enables us to determine the system level impact of different CPU and RU configurations. As an example, we have mapped an FIR filter onto different variants of the hybrid architecture. First results show that hybrid multi-context architectures have the potential to yield significant speedups. Specifically, going from a single-context to a multi-context device delivers an additional benefit. Future work includes:

- The variation of architecture parameters in order to determine their impact on the system performance.

TABLE I

CONFIGURATIONS OF THE EMBEDDED AND THE SUPERSCALAR CPU MODEL.

Architecture Parameter	Embedded CPU	Superscalar CPU
Integer units	1 ALU, 1 multiplier	4 ALU, 1 multiplier
Floating point units	1 ALU, 1 multiplier	4 ALU, 1 multiplier
L1 I-cache / D-cache	32-way 16K / 32-way 16K	1-way 16K / 4-way 16K
L2 cache	none	4-way 256K (unified)
Memory bus width / ports	32 bit / 1	64 bit / 2
IFQ / RUU / LSQ sizes	1 / 4 / 4 instructions	4 / 16 / 8 instructions
Decode / issue / commit widths	1 / 2 / 2 instructions	4 / 4 / 4 instructions
Execution order	in-order	out-of-order
Branch prediction	always not-taken	bimodal

TABLE II

PERFORMANCE RESULTS FOR FILTERING 64K SAMPLES.

CPU model	RU model	Cycles	Cycles per (tap$*$sample)	Speedup vs. embedded CPU	Speedup vs. superscalar CPU
embedded	none	91,889,988	24.6	1.0	0.3
embedded	single-context	13,470,669	3.6	6.8	2.1
embedded	multi-context	9,729,923	2.6	9.4	2.9
superscalar	none	28,109,343	7.5	3.3	1.0
superscalar	single-context	3,905,201	1.0	23.5	7.2
superscalar	multi-context	2,595,088	0.7	35.4	10.8

- The development of an area model for the reconfigurable unit in order to quantify the hardware overhead introduced by the multi-context features.
- Enhancements to the current RU model such as support for storing the state upon a context switch and integration of a dedicated RU memory port to the main memory.

REFERENCES

[1] S. Sezer, J. Heron, R. Woods, R. Turner, and A. Marshall, "Fast partial reconfi guration for FCCMs," in *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1998, pp. 318–319.

[2] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA confi gurations," in *Proc. 7th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1999, pp. 286–287.

[3] S. Hauck, "Confi guration prefetch for single context reconfi gurable coprocessors," in *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 1998, pp. 65–74.

[4] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 22–28.

[5] Xiaoping Ling and H. Amano, "WASMII: An MPLD with data-driven control on a virtual hardware," *The Journal of Supercomputing*, vol. 9, no. 3, pp. 253–276, 1995.

[6] T. Fujii, K. i. Furuta, M. Motomura, M. Nomura, M. Mizuno, K. i. Anjo, K. Wakabayashi, Y. Hirota, Y. e. Nakazawa, H. Itoh, and M. Yamashina, "A dynamically reconfi gurable logic engine with a multi-context/multi-mode unifi ed-cell architecture," in *Int. Solid-State Circuits Conf. (ISSCC), Digest of Technical Papers*, 1999, pp. 364–365.

[7] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura, "A virtual hardware system on a dynamically reconfi gurable logic device," in *Proc. 8th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2000, pp. 295–296.

[8] A. DeHon, "DPGA-coupled microprocessors: Commodity ICs for the early 21st century," in *Proc. 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, 1994, pp. 31–39.

[9] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A fi rst generation DPGA implementation," in *Canadian Workshop on Field-Programmable Devices (FPD)*, 1995, pp. 138–143.

[10] J. Faura, C. Horton, P. van Duong, J. Madrenas, M. A. Aguirre, and J. M. Insenser, "A novel mixed signal programmable device with on-chip microprocessor," in *Proc. 19th IEEE Custom Integrated Circuits Conf. (CICC)*, 1997, pp. 103–106.

[11] S. M. Scalera and J. R. Vázquez, "The design and implementation of a context switching FPGA," in *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1998, pp. 78–85.

[12] D. I. Lehn, K. Puttegowda, J. H. Park, P. M. Athanas, and M. T. Jones, "Evaluation of rapid context switching on a CSRC device," in *Proc. 2nd Int. Conf. on Engineering of Reconfi gurable Systems and Algorithms (ERSA)*. CSREA Press, 2002, pp. 209–215.

[13] Xinan Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding confi guration latency in Chameleon processors," in *Field-Programmable Logic and Applications (Proc. FPL)*, ser. LNCS, vol. 1896. Springer, 2000, pp. 29–38.

[14] J. E. Carrillo Esparza and P. Chow, "The effect of reconfi gurable units in superscalar processors," in *Proc. 9th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2001, pp. 141–150.

[15] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[16] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfi gurable coprocessor," in *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 12–21.

[17] R. Enzler, M. Platzner, C. Plessl, L. Thiele, and G. Tröster, "Reconfi gurable processors for handhelds and wearables: Application analysis," in *Proceedings of SPIE*, vol. 4525, 2001, pp. 135–146.