WOOLCANO: AN ARCHITECTURE AND TOOL FLOW FOR DYNAMIC INSTRUCTION SET EXTENSION ON XILINX VIRTEX-4 FX

Mariusz Grad and Christian Plessl

Paderborn Center for Parallel Computing, University of Paderborn
email: mariusz.grad | christian.plessl @uni-paderborn.de

ABSTRACT

In this paper, we introduce the Woolcano reconfigurable processor architecture. The architecture is based on the Xilinx Virtex-4 FX FPGA and leverages the Auxiliary Processing Unit (APU) as well as the partial reconfiguration capabilities to provide dynamically reconfigurable custom instructions. We also present a hardware tool flow that automatically translates software functions into custom instructions and a software tool flow that creates binaries using these instructions. While previous research on processors with reconfigurable functional units has been performed predominantly with simulation, the Woolcano architecture allows for exploring dynamic instruction set extension with commercially available hardware. Finally, we present a case study demonstrating a custom floating-point instruction generated with our approach, which achieves a 40x speedup over software-emulated floating-point operations and a 21% speedup over the Xilinx hardware floating-point unit.

I. INTRODUCTION

Application-specific instruction set processors (ASIPs) [1] augment the instruction set of general CPU architectures with additional instructions for accelerating specific applications. These custom instructions are tightly integrated into the CPU pipeline and perform application-specific operations on values taken from the register file. While some ASIP design tools target FPGAs as emulation platforms [2], [3] the custom instructions are fixed during the design phase and remain unchanged during runtime.

In this work we extend the concept of static custom instructions and present the *Woolcano* CPU architecture for dynamic instruction set extension. This architecture is comprised of a static part that implements the fixed instruction set and an interface for application-specific custom instructions, which can be replaced at runtime using partial reconfiguration. This capability enables novel modes of operation, such as configuring custom instructions when loading a new application, replacing custom instructions during the execution of the application, or even generating new custom instructions on-the-fly. In contrast to related works proposing new processors architectures with reconfigurable functional units, which have been evaluated primarily with simulation, our approach leverages the commercially available Xilinx Virtex-FX FPGA architecture.

II. RELATED WORK

Instruction set extension has been studied in the field of ASIPs. In particular, methods for identifying an optimal set of static custom instructions have been thoroughly investigated [4]. Architectures supporting reconfigurable custom instructions, e. g., PRISM [5], PRISC [6], DISC [7], CHIMAERA [8] or OneChip [9], have been extensively studied in the 1990's. The prime objective of these projects was to evaluate the feasibility and potential of reconfigurable functional units. Simulation-based design space exploration has been used to study speedups, the performance of CPU/RFU interfaces, the effect of instruction encoding, etc. None of these architectures has been implemented in VLSI and the limited logic capacity of early FPGAs has prevented building single-chip FPGA prototypes. Only recently a CPU with reconfigurable functional units has been commercialized with the Stretch S5 architecture [10].

III. WOOLCANO RECONFIGURABLE PROCESSOR ARCHITECTURE

The Woolcano reconfigurable processor architecture is based on the PowerPC 405 CPU core which is embedded as a hard-core in the Xilinx Virtex-4 FX FPGA. Before presenting the Woolcano architecture, we briefly introduce the APU which provides the interface between CPU and the custom instructions, as well as dynamic partial reconfiguration which is used for changing custom instructions at runtime.

III-A. APU Interface for Custom Instructions

With the Virtex-4 FX, Xilinx has introduced the Auxiliary Processor Unit (APU) in their FPGAs with embedded PowerPC cores. The APU provides a direct interface between the CPU's pipeline and the reconfigurable FPGA fabric for attaching hardware accelerators, denoted as fabric co-processor modules (FCM). For accessing an FCM, the PowerPC instruction provides three instruction classes that are decoded by the APU but executed by the FCM: user-defined instructions, floating-point instructions, and APU

Published in: *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*

load/store instructions. The Woolcano architecture uses user-defined instructions (UDI) for extending the CPU with custom instructions. The instruction set provides eight reserved opcodes (`udi0fcm-udi7fcm`) for executing UDIs. Each UDI can read two integer operands from the register file and commit the result to the register file.

III-B. Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is a configuration method supported by Xilinx Virtex FPGAs that allows for changing specific regions of the fabric at runtime, while the other parts of the fabric remain unaffected. Partial reconfiguration requires that the hardware design is partitioned into a static region and regions, that can be reconfigured at runtime. For ensuring proper communication between the static design and the reconfigurable regions, all signals need to be routed through unidirectional communication channels (bus macros). Xilinx' partial reconfiguration tool flow creates one bitstream for the static part and one bitstream for each partially reconfigurable module. After initializing the FPGA with the bitstream for the static part at startup, reconfigurable modules can be loaded through the external or internal configuration access port (ICAP) at runtime.

III-C. Woolcano Processor Architecture

The bottom part of fig. 1 shows a schematic of the Woolcano reconfigurable processor architecture. Its main components are the PowerPC core, the ICAP controller, the FCM controller, and the partial reconfiguration regions for implementing custom instructions, which we denote also as *instruction slots*. The FCM controller serves as the interface between the CPU core and the custom instructions. It forwards the inputs to the instruction slots via the operand bus and, after the custom instruction has finished computing, transfers the output back to the CPU via the result bus. The control bus is used for sending control information to the custom instructions, e. g., activation or abort signals. Bus macros are placed at the interface between the instruction slot and the operand, control, and result busses in order to enable the partial reconfiguration of the instruction slots. The number of instruction slots and the number of input operands is a configurable architecture parameter.

The user-defined instructions of the Virtex-4 FX support only two input operands and one result, which limits the amount of data on which user-defined instruction can operate. To support instructions with more inputs, we store them in an internal operand register file. The control of the register file is handled by a finite state machine. Starting from an initial state, the FCM controller waits for the notification from the APU that a UDI instruction has been decoded. If the decoded instruction is a data transfer instruction (we reserve `udi0fcm` for that purpose),

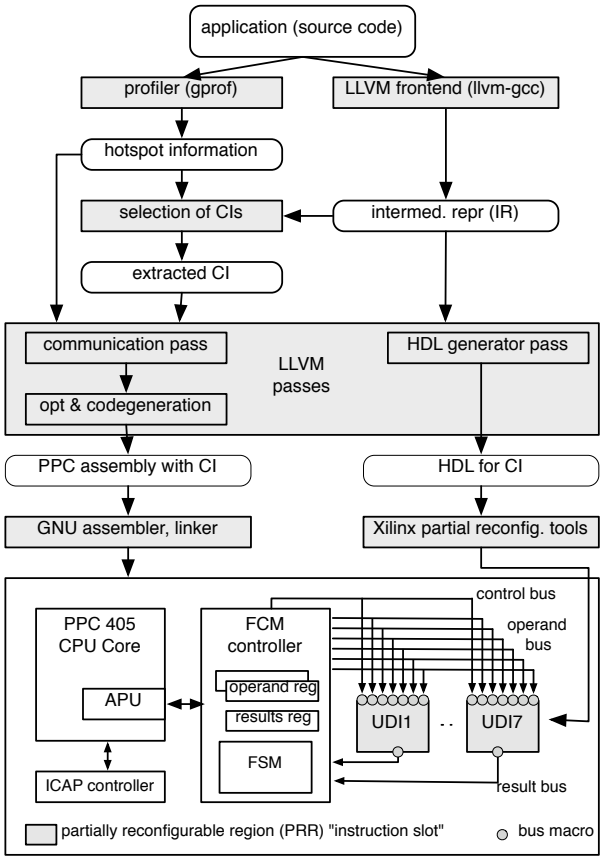

Fig. 1. Overview of the Woolcano hardware architecture and software toolflow

the controller receives and stores the operands in chunks of two operands per transfer. If the decoded instruction is a custom instruction dispatch, the instruction transfers also two operands but additionally triggers the instruction execution which causes the appropriate custom instruction to be activated via the control bus. When the result of the instruction is available, it is sent to the APU.

IV. SOFTWARE AND HARDWARE TOOL FLOW

We have developed a software and hardware tool flow for the Woolcano architecture, which builds on the LLVM compiler infrastructure [11]. LLVM's modular design and the extensible intermediate representation (IR) allow us to leverage the basic compiler infrastructure and to add our hardware and software tool flow as user-defined compiler passes.

A schematic overview of our tool flow is presented in the upper part of fig. 1. We start from an application in C code which is translated into LLVM IR using the llvm-gcc frontend. Profiling is used for identifying the most runtime intensive functions of the application. These

hotspots are analyzed and runtime critical parts that will be translated to custom instructions are identified. These custom instruction definitions are extracted from the IR and encapsulated as functions specified in LLVM IR. In our current tool flow, this extraction is performed manually, but will be automated in future. Based on these three inputs—the application compiled to IR, the hotspot information, and the IR defining the custom instructions—our tool flow generates a) an application binary using the custom instructions, and b) an HDL description of the custom instruction that can be synthesized as a partial reconfiguration for an instruction slot. These functions are added as custom transformations passes to LLVM.

IV-A. HDL Generator Pass

The HDL generator pass translates the custom instruction specification as a basic block in LLVM IR into a data path implementation. The generator ensures that translated functions do not exceed maximum number of inputs and outputs supported by the architecture. Each node of the data flow graph is mapped to an arithmetic or logical operator from a library of integer and floating-point operators.

The resulting data path is written as a VHDL netlist which is inserted into a custom instruction template that specifies the fixed interface to the FCM controller. Finally, placement constraints are applied and the custom instruction is synthesized, placed, and routed, resulting in a partial reconfiguration module that can be loaded into an instruction slot of the Woolcano architecture.

IV-B. Communication Pass

The communication pass is responsible for rewriting the LLVM IR to transfer operands to the FCM controller and to issue the custom instructions, instead of making a function call to the extracted function. We have extended the LLVM IR with two new operations (LLVM intrinsics). One to express a write to the FCM operand register file (`llvm.fpga.write(x,y)`), and another for executing custom instruction `X` (`llvm.fpga.execX(x,y)`). During code generation these intrinsics are mapped by our extended PowerPC backend to the `udi0fcm` and `udi1fcm-udi7fcm` instructions, respectively.

The communication pass is implemented as an LLVM module transformation pass, which locates the extracted function in the IR and rewrites it. The behavior of the pass is illustrated in fig. 2 with the example of a function computing the scalar product of two 3-dimensional vectors with floating-point components. First, the function to be realized as custom instruction (fig. 2a) is compiled to LLVM IR (fig. 2b) using the llvm-gcc frontend. The result of the communication pass is shown in fig. 2c. The floating-point arguments of the function are moved to the integer register file of the PowerPC using cast operations and

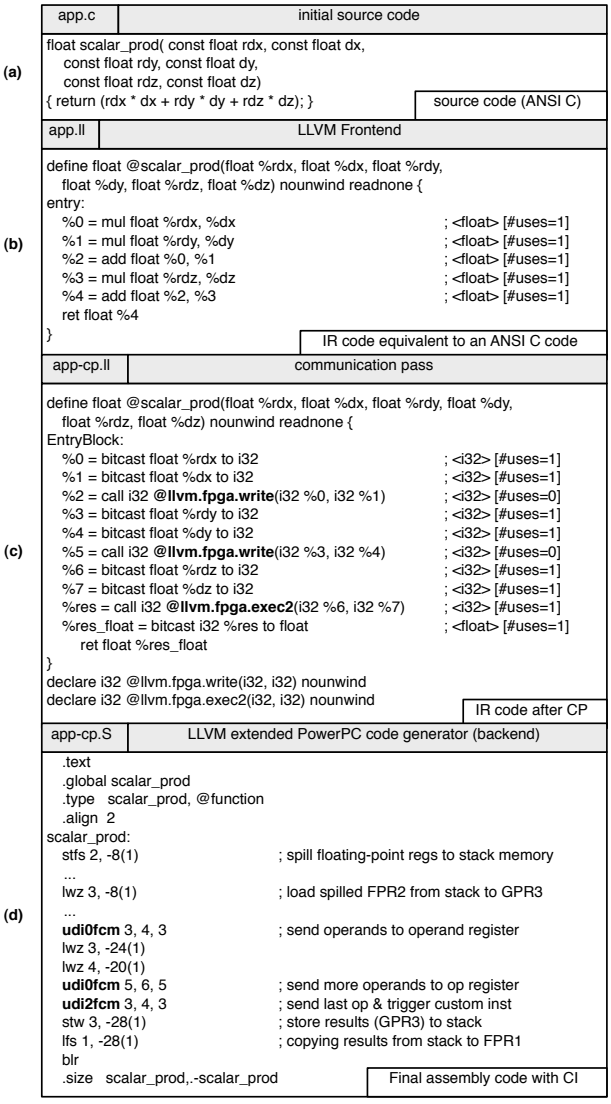

Fig. 2. Code generation with the communication pass

transferred to the FCM core using the `llvm.fpga.write` intrinsic. Then, the custom instruction is issued with `llvm.fpga.exec2`. Finally, the result committed to the register file is returned after casting it to floating-point. The assembly code generated by our extended LLVM PowerPC backend is shown in fig. 2d.

V. EXPERIMENTAL RESULTS

As a case study, we compare the performance of the floating-point scalar product operation shown in fig. 2a for three implementation variants: a) using software floating-point emulation, b) with a general APU-attached floating point FPU, and c) with a dedicated custom instruction. We have implemented the case study on a AVNET PCI Express

architecture	cycles	speedup vs. sw emulation	speedup vs. APU FPU
SW floating-point emulation	1'051'629'842	1.0	0.03
Xilinx APU FPU v3.0	31'752'386	33.12	1.00
our custom instruction	26'307'650	39.97	1.21
ICAP initialization	18'128	–	–
Reconfiguration	17'604'700	–	–

Table I. Performance of the scalar product benchmark for computing 80000 scalar products and reconfiguration time for an instruction slot of size 213'096 bytes

development board that features a Virtex-4 FX100 FPGA. We use a Woolcano architecture with one instruction slot (6 inputs, 1 outputs). The instruction slot has a capacity of 2400 slices and comprises 12 DSP48, 24 FIFO16, and 24 RAMB16 blocks. Additionally, we have attached a Xilinx FPU core v3.0 (full variant) to the APU interface, allowing us to compare the custom instruction performance and area with the general FPU.

Our hardware tool flow generated a 4 stage pipelined custom instruction with a total latency of 17 cycles between reading the inputs from the operand register file to storing the result in the result register. The FCM controller uses 77 slices. The `scalar-prod` custom instruction uses 4 registers (1 cycle latency), 3 FP multipliers (4 cycles latency) and 2 FP adders (5 cycles latency) resulting in a total area of 1879 slices (4%). The Xilinx FPU IP core consumes 1624 slices (3%) and 4 DSP48s (2%). The latency of FPU operations is not customizable (5 cycles for multiplication, 10 cycles for multiply-accumulate), which results in a total latency of 25 cycles for computing `scalar_prod`.

We run the Woolcano architecture at a frequency of 100MHz and the DDR SDRAM at a frequency of 200MHz. As required by the FPU IP core, the FPU runs at half of the system frequency. In total, the system uses 9201 slices (21%). The hardware is generated using Xilinx ISE/EDK version 9.2i with SP4, IP update 2, EAPR, and Bus Macros (24.08.2008) version 8 on Fedora Linux.

For benchmarking we have measured the time to compute 80'000 scalar products, excluding the time for system and input data initialization. The program code and data is stored in the SDRAM memory. All benchmarks have been compiled with optimization level `-O3` and are executed with enabled PowerPC caches. Table I summarizes the results for this benchmark. Compared to software floating-point emulation, the use of the FPU provides a speedup of $33.12\times$, our custom instruction provides a speedup of $39.97\times$ not including the reconfiguration time. The last column shows that our custom instruction outperforms the general FPU core by $1.21\times$.

The time needed for reconfiguring the instruction slot comprises the initialization of ICAP interface, which takes 18'128 cycles, and the reconfiguration process itself, which is dependent on the size of the instruction slot. For the setup of the case study, the reconfiguration time for the instruction slot with a configuration size of 213'096 bytes is 17'604'700 clock cycles, which corresponds to 176ms at 100MHz. This time can be reduced by more than an order of magnitude by improving the ICAP port interface [12].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented Woolcano processor architecture for dynamic instruction set extension which details our initial presentation of the architecture in [13]. In addition, we have introduced our widely automated, LLVM-based tool flow for generating custom instruction hardware and for generating software that uses these instructions. Finally, we have shown the feasibility of the approach with a case study that demonstrates how a moderately complex custom instruction achieves a 40x speedup over a pure software solution and a 21% speedup over the Xilinx standard FPU core.

In future work, we will extend our tool flow with methods for automated custom instruction identification. Additionally, we plan to evaluate the potential of dynamic instruction reconfiguration by performing case studies that change the contents of the instruction slots at runtime.

VII. REFERENCES

[1] R. Leupers and P. Ienne, Eds., *Customizable Embedded Processors: Design Technologies and Applications*, ser. Systems on Silicon. Morgan Kaufmann Publishers, Aug. 2006.

[2] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, Mar. 2000.

[3] *Mimosys Clarity product datasheet*, Mimosys, Lugano, Switzerland, Jul. 2006.

[4] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," in *Proc. Int. Conf. on Architecture of Computing Systems (ARCS)*, ser. LNCS, no. 4943. Springer, 2008, pp. 209–220.

[5] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.

[6] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. Int. Symp. on Microarchitecture*. ACM, 1994, pp. 172–180.

[7] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. Workshop on FPGAs for Custom Computing Machines (FCCM)*, 1995, pp. 99–107.

[8] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2000, pp. 225–235.

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

[10] J. M. Arnold, "S5: the architecture and development flow of a software configurable processor," in *Int. Conf. on Field Programmable Technology (ICFPT)*, Dec. 2005, pp. 121–128.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2004 Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, Mar. 2004, pp. 75–86.

[12] C. Claus, F. Muller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–7.

[13] M. Grad and C. Plessl, "Poster abstract: Woolcano – an architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Apr. 2009.