U.P.B. Sci. Bull., Series C, Vol. 78, Iss. 4, 2016 ISSN 2286-3540

A SOFTWARE-DEFINED FPGA VECTOR PROCESSOR WITH APPLICATION-AWARE RECONFIGURATION

Alexandru GHEOLBANOIU[1], Vlad POPESCU[2], Radu HOBINCU[3], and Lucian PETRICA[4]

Field-Programmable Gate Arrays (FPGAs) dominate the embedded high-performance computing application space, due to their energy efficency. Compared to software, FPGAs require more effort for application implementation. Soft processors, i.e., processors implemented in FPGA, may be utilized to reduce the implementation effort, but this sacrifices some of the FPGA intrinsic performance and energy efficency. We propose the use of a software-defined FPGA vector processor as a method to facilitate application development while providing increased performance compared to an existing soft processor design. The software-defined vector processor (SDP) structure implements only the instructions required by an application, reducing the size of the SDP and enabling increased parallelism. The FPGA may be reconfigured with different SDPs in response to changes in the characteristics of the executing application. We propose a low-latency partial reconfiguration method which overlaps application data transfer and reconfiguration. Evaluations of the SDP on a ZedBoard development board equipped with a Xilinx Zynq FPGA device demonstrate that processor customization enables a 38% increase of the maximum SDP vector size and an up to 62% reduction in reconfiguration time on a synthetic benchmark.

Keywords: SIMD, FPGA, Reconfiguration

1. Introduction

Field Programmable Gate Arrays (FPGAs) are a class of configurable circuits which enable fast and energy-efficient parallel data processing. FPGAs consist of a fine-grained network of small Look-Up Tables (LUTs), equivalent to digital circuit gates, as well as embedded memories (Block RAMs, or BRAMs), and embedded multipliers (digital signal processing blocks, or DSPs), all of which may be connected and utilized to implement complex functions. The FPGA function is encoded in the configuration of its LUTs, BRAMs, DSPs and interconnects, and does not require instruction fetching as do traditional CPUs

[1]MsC Student, DCAE Department, University "Politehnica" of Bucharest, Romania
[2]BSc Student, DCAE Department, University "Politehnica" of Bucharest, Romania
[3]Lector, DCAE Department, University "Politehnica" of Bucharest, Romania
[4]Lector, DCAE Department, University "Politehnica" of Bucharest, Romania, e-mail: `lucian.petrica@upb.ro`

or graphical processing units (GPUs). FPGAs may also process multiple data in parallel or in deep pipelines. These facts make FPGA circuits inherently more energy-efficient than CPUs.

Designing a FPGA implementation of a given function is a lengthy, labour-intensive process, a fact which is limiting the overall usefulness of FPGAs for data processing. A solution for fast FPGA implementation of functions is to utilize soft processors, i.e., processors which are implemented in FPGA [1, 2]. The advantage is that these soft processors are pre-designed and verified, and the implementation problem becomes analogous to that of software implementation. However, the soft processor must fetch and execute instructions sequentially, losing some of the advantage of FPGA implementation. To compensate, the soft processor must harness parallelism by performing operations over entire vectors of elements using a single instruction, resulting in a so-called vector processor [3]. Vector processors belong to the Single Instruction Multiple Data (SIMD) class of parallel machines, according to Flynn's taxonomy [4], and can exploit the data parallelism present in large scientific, multimedia and computer vision applications [5, 6, 7].

In this paper, we modify an existing FPGA vector processor, the ConnexArray [7], in order to make it software-defined, i.e., adaptable to the requirements of a given application. In our implementation, the application is implemented with OPINCAA [8], the ConnexArray programming environment. Custom modifications to OPINCAA enable the analysis of the resulting ConnexArray code, and the identification of unused instructions. The ConnexArray architecture itself is modified in order to enable the selective removal of instruction logic from the processor structure, without affecting the functionality of the remaining instructions, and the customization of the vector size, i.e., the number of operands processed in parallel. The unused instruction report from OPINCAA may be utilized to synthesize a customized processor, removing unused logic and instead utilizing the free-up FPGA resources to increase the size of the ConnexArray vector in order to process more data with each instruction.

As different applications may require different vector sizes, along with different vector instructions, we utilize partial dynamic FPGA reconfiguration [9, 10] to load into the FPGA a new vector processor, customized for the application currently under execution. Partial Reconfiguration is a recent method in FPGA computing to update selectively the circuitry of an FPGA, while it is still operational. We make further modifications to the ConnexArray architecture to enable partial reconfiguration and to reduce the time required for partial reconfiguration. The system is implemented on a ZedBoard development board, equipped with a Xilinx Zynq 7020 device [11]. Our evaluations focus on two applications, a finite impulse response filter (FIR), sum of absolute differences (SAD), and the sum of squared differences (SSD) distance metric, as discussion vehicles, illustrating the customization capabilities of our

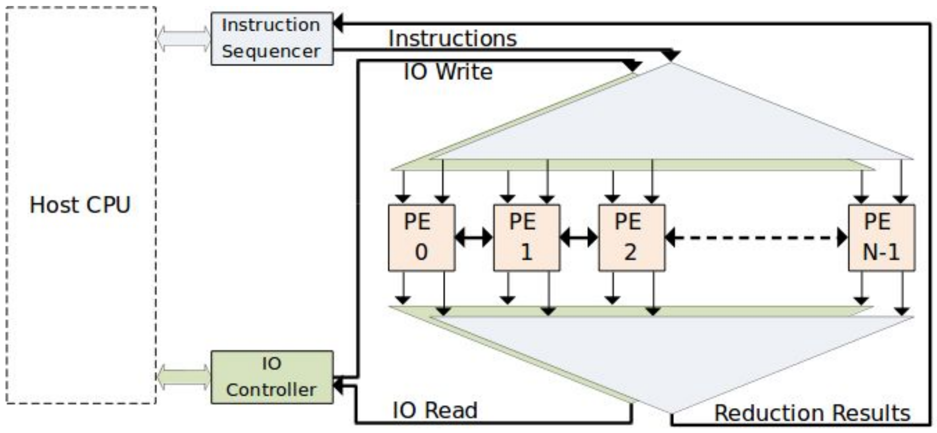


Fig. 1. The ConnexArray Architecture

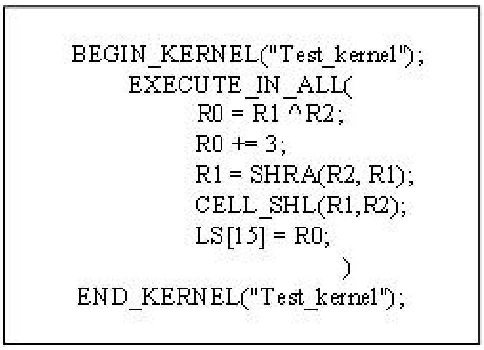
system for each one and the acceleration achievable through the combined use of customization and partial dynamic reconfiguration. Our evaluations demonstrate that the vector size may be increased by 38% through customization, and the reconfiguration time for switching between the FIR, SAD and SSD applications is reduced by 68%.

2. **Previous Work**

The ConnexArray is a data-parallel co-processor consisting of a segmented vector memory capable of storing 1024 vectors, processing logic, an instruction sequencer responsible for issuing instructions, and data input-output logic for handling read and write accesses to the vector memory. The ConnexArray architecture is presented in Figure 1. Each memory segment, called a Local Store (LS), stores one element of each of the 1024 vectors, and is tightly coupled with its own Arithmetic-Logic Unit (ALU), to form a singular processing element (PE). A host processor is responsible for setting up Direct Memory Access (DMA) transfers between main memory and the ConnexArray, in order to provide instructions and vector data to the coprocessor.

The OPINCAA programming environment is an API and library which enables a user to utilize the ConnexArray in a familiar, C++-like environment by abstracting away the complexities of DMA and automatically generating ConnexArray instructions from high level C++ code. In OPINCAA terminology, the sections of code to be executed on the ConnexArray are called kernels and are described in a special syntax, illustrated in Figure 2. OPINCAA parses the kernels at run-time and generates an instruction stream for the ConnexArray. OPINCAA also includes a ConnexArray simulator, which is useful for application development and debugging.

All ConnexArray PEs operate in parallel to process all the elements of a vector simultaneously. Additionally, the ConnexArray is equipped with an adder-tree-like network which can perform the sum reduction of a vector into



```
BEGIN_KERNEL("Test_kernel");
     EXECUTE_IN_ALL(
           R0 = R1 ^ R2;
           R0 += 3;
           R1 = SHRA(R2, R1);
           CELL_SHL(R1,R2);
           LS[15] = R0;
                        )
 END_KERNEL("Test_kernel");
```

Fig. 2. An OPINCAA Kernel

a scalar. This reduction network operates independently of the PEs and accelerates dot-product operations. An 128 PE FPGA implementation of the ConnexArray has demonstrated good energy efficiency at visual search applications based on the SSD algorithm, compared to CPUs and GPUs [7]. However, even for the algorithms evaluated in previous work, the ConnexArray structure is overly complex, and there is no mechanism in place to eliminate instruction logic if it is unutilized by an application. The previously implemented 128-PE ConnexArray occupies almost all the logic resources of a Zynq 7020 device, and the top frequency is reduced to only 100 MHz because of the high congestion of the FPGA device.

Following up on the ConnexArray previous work, in this paper we propose to modify the architecture of the ConnexArray and to enhance its programming environment OPINCAA to support the customization of the ConnexArray hardware structure to the requirements of the application. Compared to previous work on the ConnexArray, our approach reduces the size of the processor and increases its performance for a given application. However, a real-life workload consists of several applications or distinct computational kernels, each with its own characteristics, executing in sequence on the ConnexArray. Therefore, the structure of the ConnexArray in the FPGA must change accordingly, by reconfiguring the FPGA during operation. Several research groups have touched upon this subject, most notably the MOLEN [12] and RoVex [13] projects. MOLEN is a processing system consisting of a host processor and hardware accelerators, which are dynamically configured into the FPGA fabric when their functionality is required by the application. RoVex is a Very Large Instruction Word (VLIW) FPGA processor, which has a customizable instruction set and VLIW lane width, and which also adapts to the requirements of the application through FPGA dynamic reconfiguration. To date, the principal problems of dynamic reconfiguration have been the large latency and energy consumption. In previous work, processors stop executing while reconfiguration is performed, and data must be flushed out of the FPGA and back into it after reconfiguration is done, at the cost of increased energy consumption. Our innovative approach alleviates this problem by overlapping reconfiguration of one section of the processor with data transfer or execution in another section.

3. **Instruction Set Customization**

The principal design decision with regard to instruction set customization is whether to perform fine-grained customization, i.e., each instructiom may be disabled individually, or coarse grained, whereby groups of instructions are disabled together. As a matter of principle, instructions which share logic resources cannot be disabled individually, as removing the logic for one would also prevent the other from operating correctly. We therefore analyze the post-synthesis structure of the ConnexArray in order to gain more insight into the resource allocation between instructions.

The ConnexArray instruction set architecture (ISA) is illustrated in Table 1. Also listed in the table are the instruction classes, each of which comprises one or more similar instructions which share FPGA resources. For example, the Arithmetic class consists of Sub, Add along with their immediate value and carry variants. Similarly, Comparison consists of all instructions which perform comparison on operands, regardless of the type of comparison. The third column of Table 1 describes the type of FPGA resource utilized for implementing a particular instruction class: LUT, BRAM, DSP, or interconnect. Given the observed structure of the ConnexArray, the only feasible approach is to customize the ISA at the instruction class level, removing e.g. all Arithmetic instructions or all Logic instructions together.

Table 1

Instruction Classes and FPGA Resources

Instruction	Class	Resource Type
ADD SUB	Arithmetic	LUT
EQ LT ULT	Comparison	LUT
NOT OR AND XOR	Logic	LUT
MUL	Multiplication	DSP
READ WRITE	Memory	BRAM
ICSH	Inter-Cell Shift	Interconnect
RED	Reduction	LUT

The entire ConnexArray PE was re-implemented in VHDL, with each instruction class in Table 1 implemented as a separate VHDL entity (functional

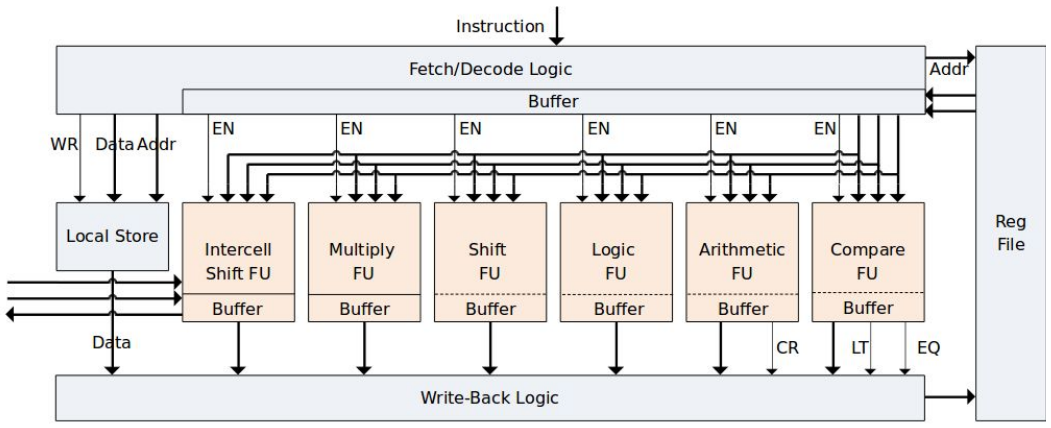


Fig. 3. The ConnexArray Processing Element

unit, or FU) and conditionally instantiated into the PE. The Memory and Reduction classes could not be made optional as their structure is necessary for input-output operations. Top-level VHDL generics dictate whether an FU is instantiated or not. The structure of the PE is illustrated in Figure 3. Operands are read from the register file under the control of the fetch logic and forwarded to the FUs, each of which delivers a result and optionally a flag to the write-back logic. The intercell shift FU also transmits and receives data to and from the adjacent PEs. If a FU is not instantiated, the value of the result signal is zero, therefore if a disabled instruction is executed, it always returns zero.

The values of the top-level generics are produced by OPINCAA after application analysis at run-time. The analysis is implemented as a two-stage process, illustrated in Figure 4. The first analysis stage occurs during kernel parsing, and produces an instruction histogram for each kernel. The second stage occurs before execution is initiated on the ConnexArray, when instructions from multiple kernels are aggregated into a single instruction stream. A weighted merge of the individual kernel histograms is performed in this stage, with the weight of each kernel determined by the number of times it is executed during the application. For analysis, the application may be executed on the OPINCAA simulator or on real hardware. The final instruction histogram indicates which instructions are never utilized, and finally, the VHDL generic values are produced by OPINCAA, applied to the ConnexArray VHDL code, and the ConnexArray is re-synthesized.

4. **Vector Processor Reconfiguration**

FPGA reconfiguration is utilized as a means of adapting the vector processor structure to the executing application. However, previous work has demonstrated significant latency associated with the FPGA reconfiguration process. This latency is caused by the transfer of the FPGA configuration

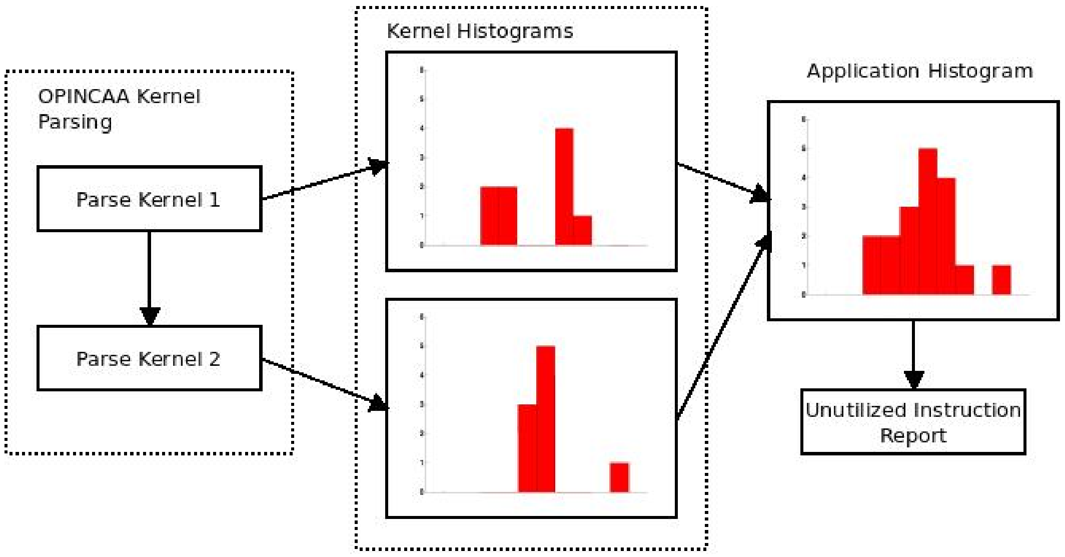


Fig. 4. Identifying Unutilized Instructions

bits, and by the transfer of application data into the processor, as a prerequisite for application execution. The analysis and minimization of the vector processor reconfiguration latency is the focal point of this section.

4.1. Quasi-Complete Reconfiguration

In the complete reconfiguration scenario, which serves as a reference, separate FPGA images are constructed through synthesis of the processor VHDL code with different values for the VHDL generics. Each FPGA image implements a customized vector processor which corresponds to an application. The customized processors however have identical instruction sequencers, input-output controllers, and bus interfaces. Therefore, a first approximation improvement on the complete reconfiguration scenario is quasi-complete reconfiguration, whereby only the PEs are reconfigured, while the remaining logic, which we shall refer to as the Static Area (SA), is not.

4.2. Partial Reconfiguration

A further improvement of the quasi-complete reconfiguration scenario is possible if we take into account the possibility of reconfiguring only half of the vector processor PEs at once. In this partial reconfiguration scenario, an application which utilizes only a small number of PEs will incur a smaller reconfiguration latency. In order to support such a scenario, modifications were made to the ConnexArray architecture allowing the isolation of each of two halves of the vector processor, hereupon called PE networks, for reconfiguration. The isolated PE network retains LS and register file data if no reconfiguration occurs, but does not execute instructions. The non-isolated PE network operates as usual.

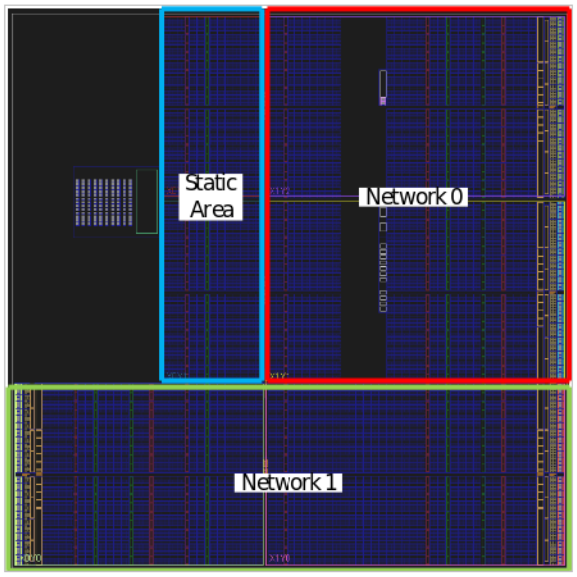


Fig. 5. SA and PE network placement on Zynq die

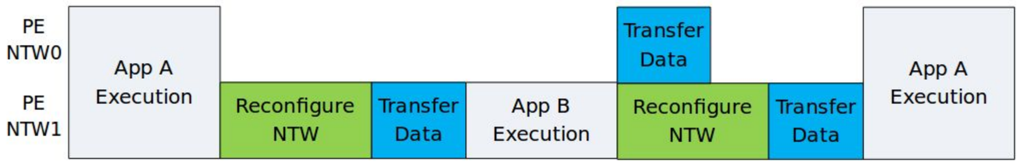


Fig. 6. Applications alternating on Vector Processor

An extra instruction is added to the ISA for activating and deactivating the isolation on each PE network. Modifications to the ConnexArray synthesis scripts are required in order to constrain the placement and routing of the FPGA such that PE networks do not overlap or share resources, which would make separate reconfiguration impossible. Figure 5 presents a Xilinx Zynq 7020 FPGA die, illustrating the SA, as well as the PE networks and their placement on the die.

An envisioned reconfiguration sequence is presented in Figure 6, where application A requires both PE networks of the vector processor for execution, while application B requires only PE network 1. Because of this, PE network 2 can remain configured for application A, and the latency of the reconfiguration is reduced. An even more favorable scenario is presented in Figure 7, where each of the applications A and B require a single PE network. Therefore, PE network 0 is configured for application A, and PE network 1 for application B. In this scenario, no reconfiguration needs to take place, as when one application is executing, the unused PE network is isolated and retains application data.

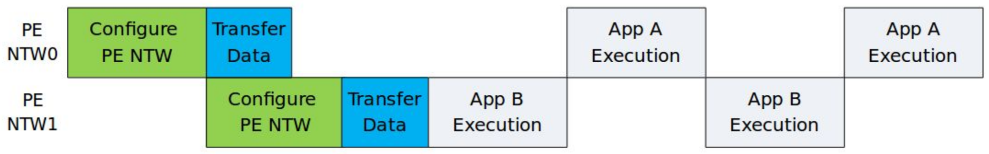


Fig. 7. Vector Processor sharing by Applications

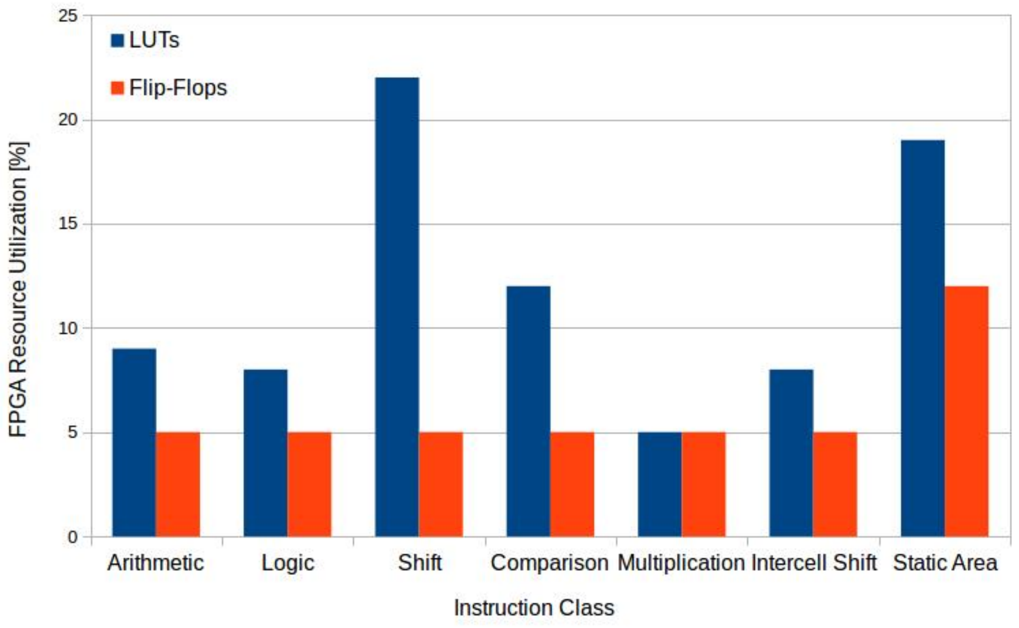


Fig. 8. Resource Utilization per Instruction Class

5. **Evaluation**

To evaluate the ISA customization and its effect on maximum ConnexArray vector size, 64 single-kernel applications were written in OPINCAA, designed to generate each possible combination of FUs. A 128-PE ConnexArray was re-customized and re-synthesized for each application utilizing Xilinx ISE System Edition 14.7, targeting the Zynq 7020 FPGA device. The resulting FPGA was image verified for correctness, and analyzed for FPGA resource utilization by inspecting the ISE reports. Figure 8 illustrates the resource utilization of the instruction classes, as percentage of the total LUTs and Flip-Flops available on the target FPGA device. The resource utilization of the static area included for reference. As demonstrated by the figure, the shift and comparison classes are the most resource-heavy. Multiplication utilizes very few LUTs and Flip-Flops, but utilizes 128 DSPs, while all other instruction classes do not utilize DSPs.

We also evaluated the maximum vector size for each processor configuration, with some of the configurations and achievable vector sizes listed in Table 2. The entire set of results could not be included due to space constraints. The average maximum vector size for all configurations is 177, a 38% increase over the full-ISA ConnexArray.

Table 2

Achievable vector size per Configuration

Mult.	Comp.	Arith.	Shift	Logic	I-C Shift	Maximum PEs
NO	YES	YES	YES	YES	YES	130
YES	YES	YES	NO	YES	YES	153
YES	NO	YES	NO	NO	NO	220
NO	NO	YES	YES	YES	NO	157

To evaluate the FPGA dynamic reconfiguration, we implemented three applications: visual search with SIFT [14] nearest-neigbor matching based on Sum of Squared Differences (SSD) and Sum of Absolute Differences (SAD), and a 64-order Finite Impulse Response (FIR) filter application. The applications require different ConnexArray configurations, as illustrated in Table 3. We analyse the reconfiguration time required for switching between these applications in the entire FPGA reconfiguration, quasi-complete reconfiguration, and partial reconfiguration scenarios.

Table 3

Processor configuration for FIR and SSD

Application	Mult.	Comp.	Arith.	Shift	Logic	I-C Shift	Required PEs
FIR	YES	YES	YES	NO	NO	YES	64
SSD	YES	YES	YES	NO	NO	NO	128
SAD	NO	YES	YES	NO	NO	NO	128

Figure 9 illustrates the ConnexArray system implemented in a Xilinx Zynq 7020 FPGA device on the Digilent ZedBoard development board, running a Linux operating system. A Xillybus DMA core [15] was utilized for transferring instructions and data between the host ARM processors on the Zynq device (the Zynq Programmable System - PS) and the ConnexArray in the Zynq FPGA (the Zynq Programmable Logic - PL). Time measurements are performed utilizing the Linux time infrastructure, with sub-millisecond accuracy. FPGA reconfiguration is achieved by transferring the configuration information, called a bitstream, from the Zynq main memory, through the AXI bus, to the configuration access port.

Table 4 lists the time required to reconfigure the entire FPGA, each PE network, and both PE networks together, respectively. The two FPGA areas reserved for PE networks 0 and 1 are of different sizes (see Figure 5) therefore

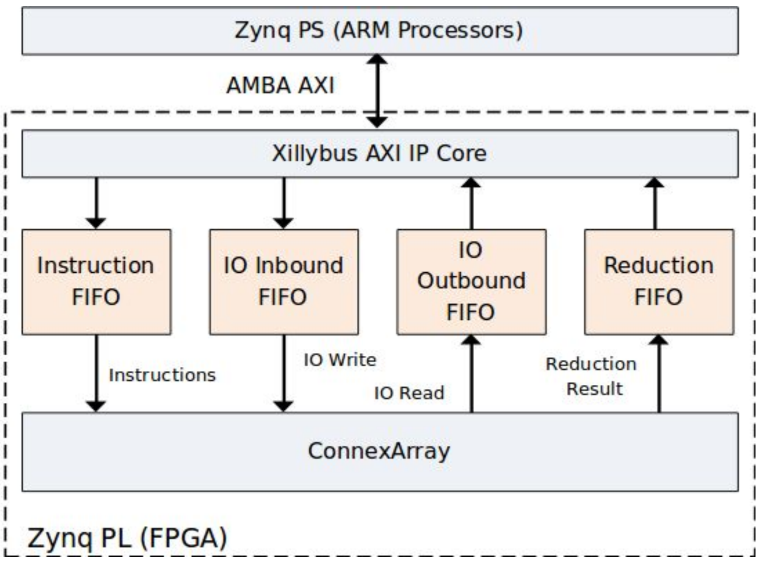


Fig. 9. The ConnexArray Zynq Implementation

the reconfiguration time is also different. Cofiguring both PE networks simultaneously corresponds to the quasi-complete reconfiguration scenario. The reconfiguration speed is limited by the configuration access port and not the AXI bus, which is utilized only to half capacity.

Table 4

FPGA Reconfiguration Performance

Scenario	Area [%]	Bitstream Size [MBytes]	Duration [ms]
Entire FPGA	100	4.1	114
PE Network 0	39	1.3	37
PE Network 1	43	1.4	39
Both PE Networks	82	2.7	76

Figure 10 illustrates the improvement attainable through using quasi-complete reconfiguration instead of complete FPGA reconfiguration, when switching between the FIR and SSD applications. The entire reconfiguration event consists of the actual FPGA reconfiguration time and the time required to fill the local store with application data, for FIR or SSD, which is 38 ms each time. Compared to the complete reconfiguration total time of 152 ms, the 114 ms total time for quasi-complete reconfiguration is a 25% improvement. Application run-time was not included in the evaluation since it may vary according to application-level latency requirements.

Partial reconfiguration is illustrated in Figure 11. We first evaluated switching between SSD and SAD. These applications each require both PE networks. Partial reconfiguration enables the overlapping of the local storage fill on PE network 0 while PE network 1 is being reconfigured. There is no

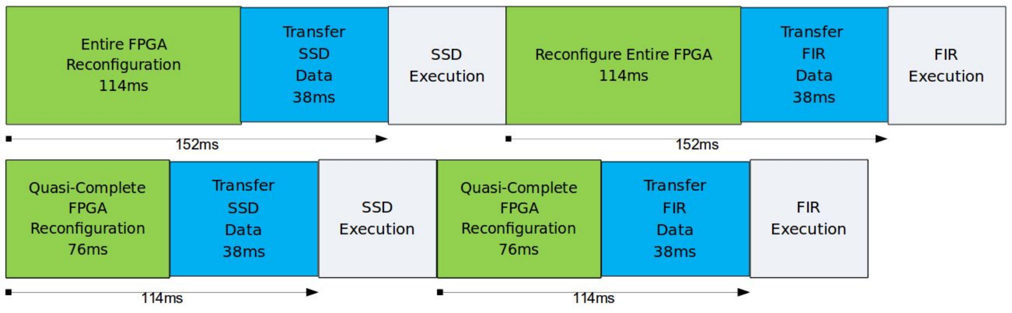


Fig. 10. Duration of Complete and Quasi-Complete Reconfiguration

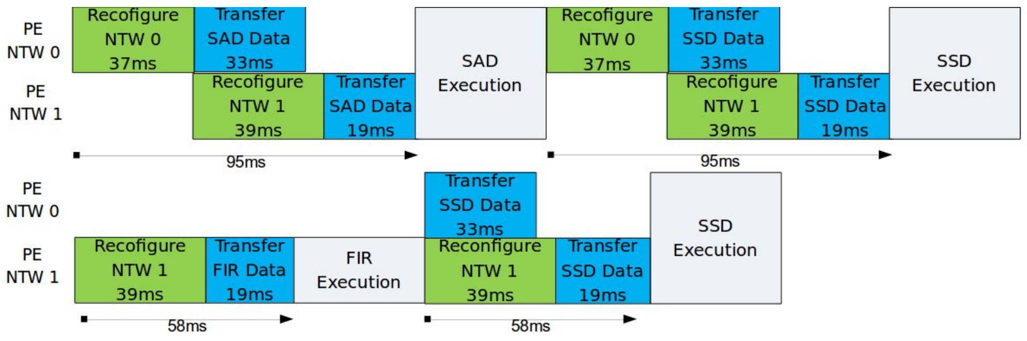


Fig. 11. Duration of Partial Reconfiguration

effect on the reconfiguration time of PE network 1 because it is limited by the configuration access port. The local storage fill takes longer because it shares AXI bandwith with the reconfiguration process, but does not add any time to the overall reconfiguration time, which is 95 ms. Compared to the entire FPGA reconfiguration scenario, this is a 37% improvement. When switching between FIR and SSD, a single PE network must be reconfigured, while the other retains its configuration (always configured for SSD). In this case, the switching time is reduced to 58 ms, a 62% improvement on the complete FPGA scenario. Table 5 summarizes the reconfiguration scenarios and the duration of each.

Table 5

Summary of Reconfiguration Times

Scenario	Total Reconf. Time [ms]	Speed-Up
Entire FPGA	152	1
Quasi-Complete	114	1.25
Partial, Both PE Networks	95	1.37
Partial, One PE Network	58	1.62

6. Conclusions

A large obstacle for the utilization of FPGAs in many computing fields is the difficulty of implementing functions on a FPGA. By utilizing soft processors, the complexity of porting applications to FPGA is greatly reduced. By utilizing vector processors and harnessing the parallelism opportunities offered by the FPGAs, much of the latent FPGA performance can be harnessed. We have proposed a software-defined soft vector processor which is capable of adapting its structure and instruction set to the executing application. We have demonstrated that up to 38% increase in parallelism may be achieved by tailoring the processor structure to the requirements of the application. We have also implemented a partial reconfiguration mechanism which enables the processor to take advantage of the requirements of the executing applications and achieve up to 62% reduction in reconfiguration time. As future avenues of research, we intend to extend the evaluations to a larger set of applications and standard parallel processing benchmarks, and to extend the configurability of the processor to the reduction network, the local store memory, and the dimensions of the register file.

REFERENCES

[1] *Xilinx Inc.*, Xilinx MicroBlaze, Online: www.xilinx.com/tools/microblaze.htm, 2014.

[2] *Altera Inc.*, Altera Nios-II, Online: www.altera.com/products/processors/overview.html, 2014.

[3] *R. Lysecky, and F. Vahid*, Design and implementation of a MicroBlaze-based warp processor, ACM Transactions on Embedded Computing Systems **8**(2009), No. 3, 22.

[4] *M.J. Flynn*, Some computer organizations and their effectiveness, IEEE Transactions on Computers, **100**(1972), No. 9, 948-960.

[5] *D. Kim, K. Kim, J.-Y. Kim, S. Lee, and H.-J. Yoo*, An 81.6 GOPS object recognition processor based on NoC and visual image processing memory, IEEE Custom Integrated Circuits Conference, 2007, 443446.

[6] *K. Asanovic, B. Kingsbury, B. Irissou, J. Beck, and J. Wawrzynek*, T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines, IEEE European Solid-State Circuits Conference, 1996, 344-347.

[7] *C. Bira, R. Hobincu, L. Petrica, V. Codreanu, and S. Cotofana*, Energy-Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search, Advances in Information Science and Applications, **2**(2014), 432-437.

[8] *C. Bira, L. Petrica, and R. Hobincu*, OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators, Romanian Journal of Information Science and Technology, **16**(2013), No. 4, 336-350.

[9] *C. Kao*, Benefits of partial reconfiguration, Xcell Journal, **55**(2005), 65-67.

[10] *E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour*, Dynamic hardware plugins in an FPGA with partial run-time reconfiguration, ACM Design Automation Conference, 2002, 343-348.

[11] *L. H. Crockett*, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc, Strathclyde Academic Media, 2014.

[12] *S. Vassiliadis, S. Wong, and S. Cotofana*, The MOLEN -coded processor, Field-Programmable Logic and Applications, 2001, 275-285.

[13] *S. Wong, T. Van As, and G. Brown* Ro-VEX: A reconfigurable and extensible softcore VLIW processor, IEEE International Conference on Technology, 2008, 369-372.

[14] *D. Lowe*, Object recognition from local scale-invariant features, IEEE International Conference on Computer Vision, 1999, 1150-1157.

[15] *Xillybus* Xillybus DMA IP, Online: http://xillybus.com/, 2015