An Out-of-Order Superscalar Processor on FPGA: The ReOrder Buffer Design

Mathieu Rosière, Jean-lou Desbarbieux, Nathalie Drach, Franck Wajsbürt
University of Pierre et Marie Curie – UPMC (Paris 6)
LIP6 - France

***Abstract*—Embedded systems based on FPGA (*Field-Programmable Gate Arrays*) must exhibit more performance for new applications. However, no high-performance superscalar soft processor is available on the FPGA, because the superscalar architecture is not suitable for FPGAs. High-performance superscalar processors execute instructions out-of-order and it is necessary to re-order instructions after execution. This task is performed by the ROB (*ReOrder Buffer*) that uses usually multi-ports RAM, but only two-port buffers are available in FPGA. In this work, we propose a FPGA friendly ROB (*ReOrder Buffer*) architecture using only 2 ports RAM called a *multi-bank* ROB architecture. The ROB is the main and more complex structure in an out-of-order superscalar processor. Depending on processor architecture parameters, the FPGA implementation of our ROB compared to a classic architecture, requires 5 to 7 times less registers, 1.5 to 8.3 times less logic gates and 2.6 to 32 times less RAM blocks.**

Keywords-component; superscalar processor; performance; out-of-order execution; FPGA; ReOrder Buffer

I. INTRODUCTION

The main FPGA (*Field-Programmable Gate Arrays*) target is the embedded applications (multimedia, telephony, cryptography, …) and these applications require more and more performance. The available soft processors (a soft processor is a processor implementation on a FPGA as the MicroBlaze [1], OpenRISC 1200 [2], …) have a scalar architecture and thus cannot execute more than one instruction per cycle.

Superscalar processors [3] execute several instructions per cycle. An aggressive superscalar architecture design (i.e. out-of-order execution) uses complex structures such as associative arrays or structures with multiple writing and multiple reading ports. These complex structures allow to extract dynamically the instruction parallelism, but are difficult and expensive to implement on FPGA. For example, the synthesis of a Nehalem Intel core [4] (without second level cache) needs 5 FPGAs i.e. 760k LUTs of the 900k LUTs available. The instruction parallelism extraction is more efficient in hardware (as in superscalar processors) than in software (as in VLIW processors).

In order to implement a superscalar processor on FPGA, the designer must simplify these complex structures by reducing especially the number of simultaneous accesses. For example, clustering the micro-architecture [5] can reduce the number of ports on the register file and the reservation station (a component of the out-of-order execution block) for each cluster. Another design modification is to sequentially access to complex structures. For example, the limitation of one load/store instruction per cycle can simplify the load/store unit and the data cache design. But these modifications may decrease performance.

For our architecture studies, we use the Morpheo processor generator [6]. Morpheo can simulate an out-of-order superscalar processor with configurable architectures (for example, configurable coarse-grain as superscalar degree and configurable fine-grain as ReOrder Buffer (ROB) size). It supports the integer subset of OpenRISC 1000 instructions set [2]. To target the reconfigurable logic, the Morpheo tool includes a VHDL description generator.

In this paper, we focus on the ReOrder Buffer (ROB) architecture. We present the *classic* ROB architecture for ASIC design, and we show that this *classic* architecture uses FPGA resources in a prohibitive way. Then, we propose a FPGA friendly ROB implementation taking into account the characteristics of FPGAs with low performance decreasing. Section II describes the *classic* ROB architecture and section III presents our ROB implementation (*multi-bank* architecture) on FPGA. Then, section IV shows the experimental context and the design evaluation results (performance and area). Section V presents related works. Finally, section VI summarizes our main results.

II. REORDER BUFFER

A. Superscalar processor

A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions. It checks for data dependencies between instructions at run time (versus software checking at compile time). It is found in the general-purpose and high-performance systems.

Superscalar processor architecture with an out-of-order execution is composed of three blocks: front end block, out-of-order engine block and execution loop block (see Figure 1).

The front end block fetches and decodes simultaneously several instructions (a instructions packet) from the instructions cache (icache). The current address of instructions packet is sent to the branch prediction unit (the outcomes of

978-3-9810801-8-6/DATE12/©2012 EDAA

conditional branch instructions are predicted in advance to ensure uninterrupted stream of executed instructions).
The out-of-order engine block manages the instructions. The instruction registers are renamed to remove the false dependencies (Write-after-Write and Write-after-Read). Instructions are initiated for execution in parallel based on the availability of operand data, rather than their original program sequence. This is referred to as *dynamic instruction scheduling*. After register renaming, the execute loop block (included execution units) can execute instructions out-of-order. The ReOrder Buffer (ROB) keeps the program order, the instructions are inserted in the ROB after the register renaming and come back (from execute loop) out-of-order. Upon completion instruction results are re-sequenced in the original order. Also, the instructions on the top of ReOrder Buffer can update the execution context (to preserve the program order). Execution loop block reads operands for each instruction, executes the operation and writes the result. The load-store unit manages the memory accesses.

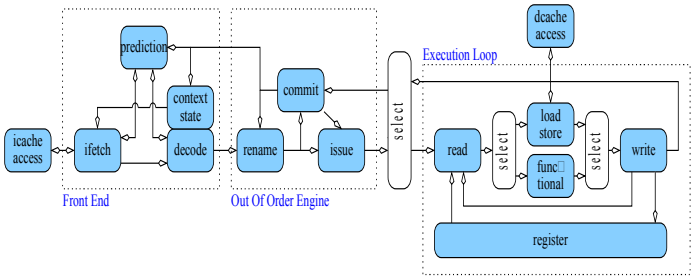

Figure 1. Superscalar functionnal blocks

B. ROB description

With the register renaming, the ReOrder Buffer is the main structure of the out-of-order engine. Figure 2 shows the ROB architecture and its interface with architecture blocks. It consists of a memory part to save instruction information and a combinatory part to update the execution context (registers and memory). The ROB communicates with the front end block, the rename logic of out-of-order engine block and the execution unit of execution loop block. It has five interfaces (one is optional):

- *Insert (number of ports is I):* instruction insertion in the ROB. After the register renaming, the instructions are stored in the ROB in the program sequential order.
- *Execute (E ports):* instructions back to the execute unit notify their termination at the ROB.
- *Commit (C ports):* instruction validation. The execution context (registers and memory) is modified if no previous instruction gives an exception or a branch prediction failure.
- *Window (W ports):* oldest instructions in the ROB. The *C* instructions are selected among the oldest *W* instructions in the ROB. These *W* instructions are called the selection window.
- *Read Operand* (optional interface, see below)(*O ports*): if the physical registers are embedded in the ROB, this interface gives the data at the consumer instruction.

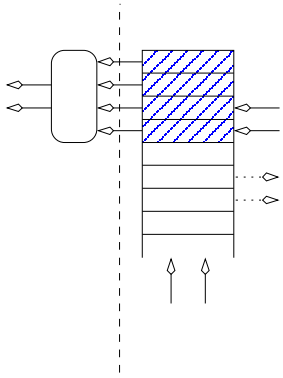

Figure 2. ReOrder Buffer (ROB) Architecture

The ReOrder Buffer architecture depends on the register renaming schemes (register renaming schemes and the location of the physical registers are presented in [7]). One scheme puts the physical registers in the ReOrder Buffer. So operand reading must be done from the register bank as well as from the ROB. Another scheme reduces the ROB complexity by combining the additional physical registers for renaming and the logical registers in the register file (see Pentium 4 [8]). The Figure 2 shows five ROB interfaces, but in our work like in Pentium 4, the physical registers are separated from the ROB, then only four interfaces are needed.

C. Information saved in the ROB

The information to update the execution context (registers and memory) are stored in the ROB. In this subsection, we list the fields for the ROB entries:

- *depth*: the number of branches before the instruction (number of predicted branches).
- *type*: the type of instruction (e.g. branch, memory access).
- *register:* this field includes three destination registers: logical register number, the new physical register number (after renaming) and the old physical register number (before renaming).
- *exception:* if the instruction in the top of ROB (the oldest instruction) makes an exception, the processor jumps to the exception handler.
- *address*: destination address of an indirect jump. It allows to test if the computed destination address is similar to the predicted destination address.
- *lsu_ptr*: this pointer indexes the store instruction in the load-store unit because the store instruction accesses at the data cache when there is no previous instruction (store instruction is in the top of ROB).
- *state:* instruction state (e.g. being executed, waiting to be validated).

Table I shows the interfaces interacting with the different fields of a ROB entry.

TABLE I. ACCESS TO THE FIELDS OF A ROB ENTRY

Fields	Write	Read	Category
state	I+C+E	W+E	state
depth	I	W	info-selection
type	I	W	info-selection
register	I	C	info-update
lsu_ptr	I	C	info-update
exception	E	C	result
address	E	C	result

Instead of implementing the ROB with one register file (with W+E read ports and I+W+E write ports) containing all fields per entry, we can group fields into four categories and implement ROB with one register file per category. Indeed, all fields of a ROB entry are not accessed at the same time in the pipeline. These categories (and implementation with several banks of registers) reduce the number of ports for a ROB entry. The last column of the Table I summarizes these four categories (*state*, *info-selection*, *info-update*, *result*).

In this section, we describe two ReOrder Buffer architectures: the first architecture concerns an ASIC implementation and it is modified for FPGA implementation. The second architecture is specific and optimized for a FPGA implementation.

D. *Like ASIC architecture : classic ROB architecture*

The memory part of the ROB needs multi-ported memories to implement each category, but the available RAMs block in FPGA typically has only two ports. To use FPGA's macro blocks (R_{fpga} read ports and W_{fpga} write ports), a method is to emulate complex memories with simple memories [4], [9]: to have multiple read ports (R_{RAM} ports), we duplicate the RAM into R_{RAM}/R_{fpga} banks, each bank having at most R_{fpga} read ports and to have multiple write ports (W_{RAM} ports), we duplicate the RAM into W_{RAM}/W_{fpga} banks, each bank having at most W_{fpga} write ports.

With this implementation, the register file isn't coherent. To solve this problem, an additional array, the Live Value Table (LVT) [9], keeps for each register the bank number with the most recently value. For each read port, a multiplexer selects the correct bank.

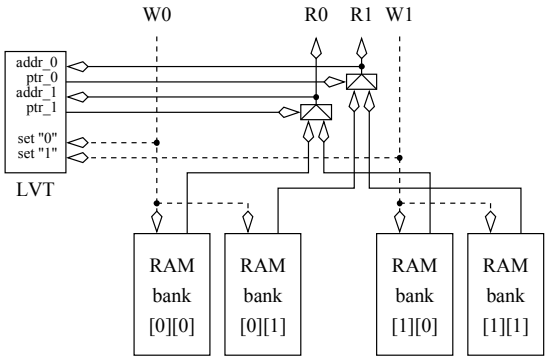

Figure 3. Register file using dual port RAMs

With multiple R_{RAM} read and W_{RAM} write ports, the RAM must be duplicated in $(W_{RAM}*R_{RAM})/(W_{fpga}*R_{fpga})$ banks. The Figure 3 is an example of register file with 2 write ports and 2 read ports using dual port RAMs. In this article, we call *classic* ROB architecture, the ReOrder Buffer using LVT register file.

E. *FPGA friendly architecture : multi-bank ROB architecture*

In this part, we describe an architecture called *multi-bank* architecture that can be implemented easier in FPGA. The principle is the division of the ROB in *N* banks, where $N \geq max(I,E,W,C)$. We assume that each bank is connected to four interfaces (*insert, execute, window, commit*).

Figure 4 shows the ROB multi-bank implementation where each bank contains all ROB field categories. The main interest of this method is dual-ported memories of *info-selection*, *info-update* and *result* sub-banks.

When an instruction saves information in ReOrder Buffer (by the Insert interface), an id is returned (the id is the bank number and the entry number in the bank). This id is used to index the ROB when the instructions come back from the execute unit.

However, the instructions come back from the execution units out-of-order, so many instructions from *execute* interface can access at the same sub-bank *result*. To maintain dual-ported *result* sub-bank, we consider that each bank can be accessed by only one instruction from the *execute* interface simultaneous. In case of conflicts (more than one instruction needs to access the same bank), a static arbitration allows to select the most prior instruction and the other instructions delayed and selected later by the ROB.

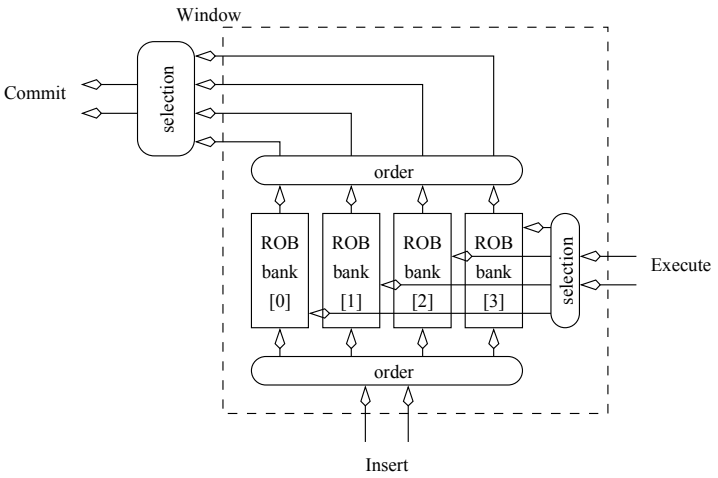

Figure 4. ROB multi-bank architecture

Each bank is managed as a FIFO (also it maintains insertion order) and can push or pop one instruction per cycle, banks must be accessed per cycle. Also, to insert or extract many instructions per cycle, several banks must be accessed per cycle. To select the correct banks and to preserve the software sequential order, we add two combinatory units to push (insert)

and pop (commit) instructions in the ROB with round robin arbitration.

III. RESULTS

In this section, we present the synthesis of ReOrder Buffer (ROB) on a Virtex 5 and the performance of the *classic* ROB architecture and the *multi-bank* ROB architecture.

A. Experimental context

We compare the ROB implementation from ASIC (*classic* ROB architecture) and our implementation (*multi-bank* ROB architecture). As the combinatory part is similar for the two architectures, we study only the queue implementation for these two architectures. For this, we implement two ROB architectures on a Virtex 5 *xc5vlx330* with speed grade -1 FPGA and with the Xilinx ISE 9.1i tool. As multi-bank implementation can generate access conflicts to the ROB banks, we analyze this impact on different benchmarks.

We have selected small benchmarks from MiBench suite [10]: basicmath (simple mathematical computation), bitcount (Bit manipulation), qsort (quick sort algorithm), susan (image recognition), dijkstra (find the shortest path), stringsearch (Insensitive comparison algorithm) and sha (secure hash algorithm) and benchmarks from SPECINT2000 [11]: gzip, bzip2 (compression), vpr (FPGA circuit placement and routing), mcf (combinatorial optimization) and twolf (place and route simulator).

For this study, we use the Morpheo processor generator [6]. It supports the integer subset of OpenRISC 1000 instructions set [2]. To target the reconfigurable logic, the Morpheo tool includes a VHDL description generator. In this study, the simulated processor is a classic 32-bit architecture, 4-way superscalar with out-of-order execution, mono-context processor and 256 physical registers (for 32 logical registers). To analyze in detail the conflict impact for instruction accesses in the ROB banks, the cache hierarchy is perfect (the access memory are performed in one cycle).

TABLE II. CATEGORY SIZES

Category	Size (bits)	
info-selection	8	(depth = 3, type = 5)
info-update	34	(register = 22, lsu_ptr = 12)
state	5	
result	37	(address = 32, exception = 5)

The ROB size ranges from 128 to 512 entries and the window size from 4 to 16 entries. A total of 9 configurations are used to evaluate the both implementations. On Table III and Figures 5, 6, 7 and 8, the 9 configurations are noted *ROB Size - Selection Window Size*. Table II shows the number of bits required for each category of ROB.

B. FPGA synthesis

We present the synthesis results of two ROB implementations (*classic* and *multi-bank*) in the Table III.

TABLE III. FPGA SYNTHESIS RESULTS

Configuration	Slice Registers	Slice LUTs		critical path (ns)
		Logic	Memory	
	classic			
128 - 4	1280	9139	5056	
128 - 8	1280	14068	5568	
128 - 16	1792	27193	6592	
256 - 4	2560	18691	10112	5.946
256 - 8	2560	29255	11136	8.680
256 - 16	3584	57425	13184	16.613
512 - 4	5120	42017	20224	
512 - 8	5120	64890	22272	
512 - 16	7168	125990	26368	
	multi_bank			
128 - 4	256	2828	632	
128 - 8	256	6464	1264	
128 - 16	256	17512	2528	
256 - 4	512	3580	632	4.413
256 - 8	512	7272	1264	7.190
256 - 16	512	19096	2528	14.838
512 - 4	1024	5036	1264	
512 - 8	1024	8808	2528	
512 - 16	1024	20840	5056	
Total available	207360	207360	54720	

Figure 5 shows the FPGA use for the *classic* implementation. The FPGA use is defined by the percentage of slice registers, the number of LUTs used as logic and the number of LUTs used as memory (RAM block). The *classic* implementation duplicates the memory banks to use blocks of RAM. The number of banks is the number of ports for reading and writing. *info-selection* category depends on the selection window (*W*) and the insert interface (*I*). With 128 entries ROB and window size of 4, the *info-selection* is duplicated in 16 banks (4 read and 4 write ports), for a total of 512 RAM blocks (each RAM block can be configured as 64Kb x1 to 1Kb x36). With the window size of 16, the *info-selection* is duplicated in 64 banks (16 read and 4 write ports), for a total of 2048 RAM blocks.

The LVT array is implemented with slice registers (because it is a multi-ported RAM). With a 128 entries ROB, field category *info-selection*, *info-update* and *result* are accessed with 4 write ports, also the LVT array of each category needs 256 registers.

Logic is used to implement LVT array, bank selection and state array. The 512 entries ROB with 16 entries selection window needs 126k LUTs (used as logic). This corresponds to 60% of FPGA resources.

For *multi-bank* implementation, the FPGA use is shown in Figure 6. The slice LUTs used as logics is mainly used for the management of sequential program order (routing network tagged as *order* in the Figure 4). The number of banks is proportional to the superscalar degree (the number of *insert*, *commit* and *update* ports) and the window size. Also, when the number of banks increases, the complexity of the management of sequential program order increases.

We can observe that for the 128-entries and 256-entries ROB, the total of RAM blocks is the same, because the RAM block used is respectively RAM32Dx1D (Dual port RAM with 32x1 bit RAM) and RAM64Dx1D. Each RAM occupies 2

LUTs. The 512-entries ROB needs the RAM128Dx1D block and requires 4 LUTs.

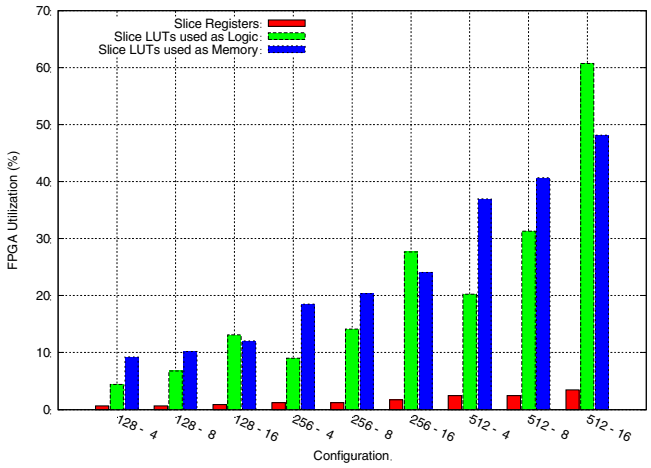

Figure 5. FPGA use of classic ROB architecture

To compare the both architectures, we compute the ratio FPGA use for the *multi-bank* implementation on the FPGA use for the *classic* implementation. The Figure 7 shows these results.

The general trend is that the *multi-bank* implementation needs less FPGA resources that the *classic* implementation. More the ROB size increases, more the gain is high, but it is decreased when the selection window size increases, because the cost of the routing network increases too.

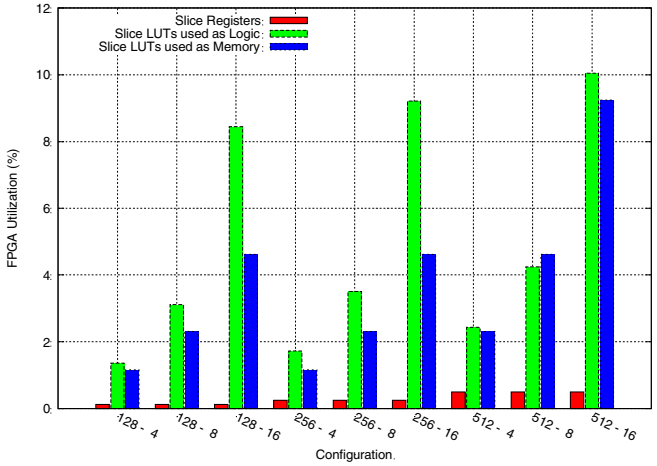

Figure 6. FPGA use of multi-bank ROB architecture

The *multi-bank* implementation reduces the number of registers between 80% and 85%, the number of LUTs used as logic between 35% and 88% and the number of LUTs used as memory between 61% and 97%.

In the Table III, we list the critical paths for the 256 entries ROB. The critical path values are almost the same for the both implementations, just a little lower for the multi-bank implementation.

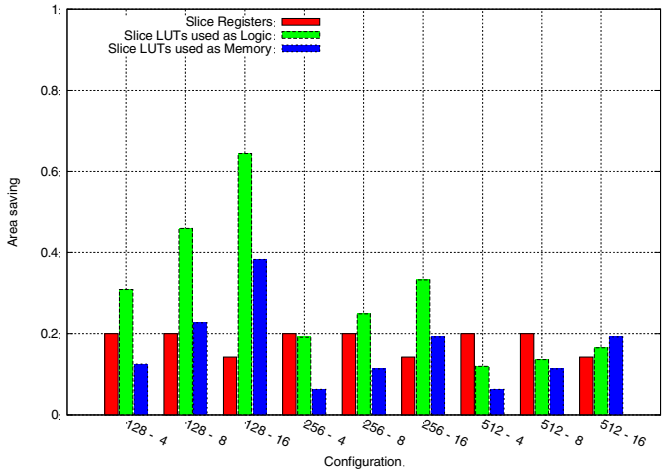

Figure 7. Ratio of the FPGA use of multi-bank ROB architecture on classic ROB architecture

C. *Performance*

Because of minor differences in critical path values, IPC can be used to compare performance. The gain area for multi-bank implementation is obtained at the expense of performance (expressed in instruction per cycle (IPC)).

So, to use FPGA macro block without register duplication, we have created conflicts between instructions accessing the same bank from *execute* interface.

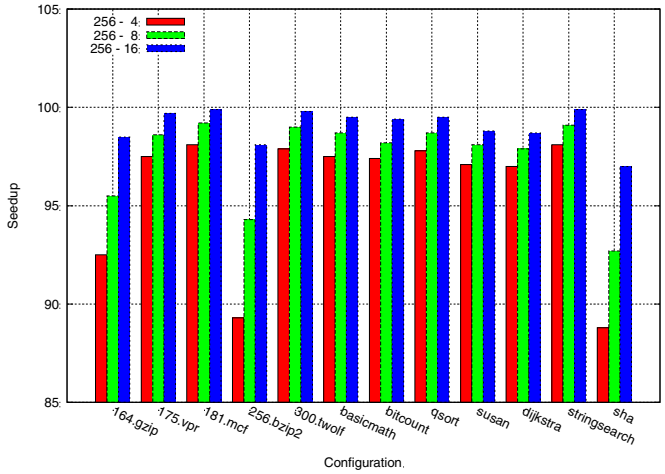

Figure 8. Ratio of the IPC of *multi-bank* architecture on *classic* architecture

The performance decreasing depends on the number of conflicts, and this number depends on the processor activities. More the applications issue instructions per cycle, more conflict accesses can occur. To measure the performance loss with the proposed architecture, we simulate the benchmarks with the 9 configurations and with both implementations.

We simulate 500 millions instructions of each benchmark and we compute the ratio IPC with *classic* implementation on IPC with *multi-bank* implementation.

Increasing the ROB size has a minor impact on the number of the conflicts, because the number of banks is unchanged. But the number of instructions sends to the execute unit increases with the ROB size. This number is limited by the available application ILP (Instruction Level Parallelism).
So, with a 128-entry ROB and 4 banks, the IPC of *multi-bank* implementation is 0.960 times the IPC of *classic* implementation, and with a 256-entry, the ratio is 0.957.

The Figure 8 shows the benchmarks results for 256 entries ROB. The applications with high instruction level parallelism (gzip, bzip2 and sha) have the most performance decreasing: with 4 banks, the performance is reduced by 10% on bzip2 benchmark compared to *classic* implementation.

Increase the number of banks reduces the probability that two instructions access at the same bank simultaneous, also the IPC decreases. We observe that with the 128-entry ROB, the IPC decreasing for 8 banks implementation is 2.2% and for 16 banks implementation, the decreasing is close to zero (0.7%)

IV. Related Works

To our knowledge, few works have been published about the FPGA implementation of ROB to execute several instructions per cycle.

The project SCOORE [12] concerns a high-performance processor on FPGA. It is a four degrees out-of-order superscalar processor. This work is still in progress and they plan a total of 117k LUTs and 25k for the ROB.

An equivalent topic is the implementation of a commercial processor on FPGA. The synthesis of high performance out-of-order processor Intel Nehalem requires 5 high density integration FPGAs [4] for a total of 760k LUTs and the most critical unit is the out-of-order Engine (ROB, renaming, …). In the same article, the authors show that at the same frequency, Nehalem Intel is by 1,8 to 3,9 more efficient in IPMC than an Atom processor. Yet Intel Atom processor is an "in order" execution superscalar successfully implemented on one Virtex 5 [13] that is the quarter of Nehalem LUTs.

Other works target a specific unit in a superscalar processor. To simplify the out-of-order Engine, in [14], the authors introduced a register renaming architecture requiring seven times less logical blocks than a classic architecture for an IPC decreasing of 3%. However, this technique can be used only with one renamed instruction per cycle.

The LVT used multi-ported memories is presented in [9]. This architecture uses FPGA RAM blocks and compared to a LUT based design, is better in area and speed.

V. Conclusion

In this work, we have synthesized a ROB with ASIC architecture (*classic* implementation). The FPGA use is very high: a 512 entries ROB with 16 instructions in the selection window uses 60% LUTs of a Virtex 5 *xc5vlx330*.
We have introduced a new ROB implementation based on multiple internal bank design. Each bank is a 2-port RAM and can be easily implemented in FPGA.

Compared to the *classic* ROB architecture, our *multi-bank* ROB architecture requires 5x to 7x less registers, 1.5x to 8.3x less logic and 2.6x to 32x less RAM blocks. In our measurements, the frequency of our *multi-bank* architecture is equivalent to the *classic* architecture and the IPC (*Instructions Per Cycle*) decreasing, at most by 10%, depends on the selection window: less of 10% with a small window and close to 0 with a large window.
The implementation of an out-of-order superscalar processor on FPGA, including complex structures like ROB, can be achieved with minor design modifications. Thanks to our multi-bank approach, an out-of-order superscalar processor can be effectively implemented on FPGA.

References

[1] XILINX, "MicroBlaze Processor Reference Guide", 2002.

[2] OpenCores, "OR1200 OpenRISC processor", http://opencores.org/openrisc/?or1200, 2009.

[3] J.E. Smith, G.S. Sohi, "The microarchitecture of superscalar processors", Proceedings of the IEEE, 83(12), pp. 1609-1624, 1995.

[4] G. Schelle and al., "Intel nehalem processor core made FPGA synthesizable", Proceeding of the ACM International Symposium on Field Programmable Gate Arrays, pp. 3-12, 2010.

[5] S. Palacharla and al., "Complexity-Effective Superscalar Processors", Proceedings of the International Symposium on Computer Architecture, pp. 206-218, 1997.

[6] M. Rosière, "MORPHEO : Open processor, high performance, parameterizable and perennial to a trusted platform", Phd University Pierre et Marie Curie, France, 2010.

[7] D. Sima, B. Polytech, "The design space of register renaming techniques", Journal of Micro, IEEE, 20(5), pp. 70-83, 2000.

[8] G. Hinton and al., "The microarchitecture of the Pentium 4 processor", Intel Technology Journal, 1, 2001.

[9] C.E. Laforest and al., "Efficient multi-ported memories for FPGAs", Proceeding of the ACM International Symposium on Field Programmable Gate Arrays, pp. 41-50, 2010.

[10] M.R. Guthais and al., "MiBench: A free, commercially representative embedded benchmark suite", Proceedings of the IEEE Annual Workshop on Workload, pp. 3-14, 2011.

[11] J.L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium", IEEE Computer Journal, 33(7), pp. 28-35, 2000.

[12] F.J. Mesa-Martinez and al., "SCOORE: Santa Cruz out-of-order RISC engine, FPGA design issues", Proceeding of the Workshop on Architectural Rsearch Prototyping, pp. 61-70, 2006.

[13] P.H. Wand and al., "Intel atom processor core made FPGA-synthesizable", Proceeding of the ACM International Symposium on Field Programmable Gate Arrays, pp. 209-218, 2009.

[14] K. Aasaraai and A. Moshovos, "Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming", Proceedings oh the Field-Programmable Logic and Applications, pp. 79-85, 2009.