An Architecture & Mechanism for Supporting Speculative Execution of a Context-full Reconfigurable Function Unit

Tao Wang, Zhihong Yu, Yuan Liu, Peng Li, Dong Liu, Joel S. Emer
wangtao@ieee.org, {zhihong.yu, yuan.y.liu, peng.p.li, dong.liu, joel.emer}@intel.com
Intel Corporation

ABSTRACT
Recently researchers have shown interest in integrating Reconfigurable logic into conventional processors as a Reconfigurable Function Unit (*RFU*). A context-full RFU supports holding intermediate results inside itself, which eliminates some data movement overheads and has some other benefits. Most contemporary processors support out-of-order execution and speculation. When a context-full RFU is integrated into a speculative processor, if a speculative RFU instruction has modified the RFU context but cannot be committed in the end, the RFU context must be recovered. Traditional mechanisms to handle speculative execution of instructions cannot effectively address this issue. Because of the design complexity, previous proposals did not support context-full RFUs in speculative processors. In this paper, we propose an architecture & mechanism for supporting speculative execution of a context-full RFU in in-order issue, out-of-order execution processors. It does not require too much extra space for the RFU context storage and the performance penalty shown to be low in practice.

1. INTRODUCTION
Reconfigurable Logic (*RL*) has the capability of providing custom operations and has more flexibility than fix-function logic. Then compared to conventional fixed function ISAs, an ISA augmented with RL has the potential to provide much better performance for a variety of computations. Since the 1990s [1], researchers have explored integrating RL into conventional processors as a reconfigurable function unit (*RFU*) to get both a performance gain over conventional fixed-function ISAs and faster time-to-feature than fixed-function logic, as is illustrated in figure 1.

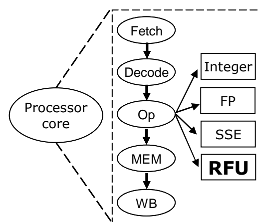

Figure 1: RFU in processor

Different kinds of RFUs can either support the notion of *context* or not. Context is a portion of the local states of an RFU that can be observed by the components outside the RFU. A simple example of context is the accumulated sum inside an RFU when the RFU provides a vector sum operation. In our definition, an RFU is called *context-full RFU* if it supports such an internal context.

A context-full RFU has at least three advantages: 1) it eliminates some data movement overhead by saving intermediate results into the RFU context; 2) it enables a way of handling more and/or wider inputs/outputs than the standard operands of instructions; 3) RFU context can serve as a look-up table for various computations, for example, as a DFA (Deterministic Finite Automaton) state transition table. We believe that the advantages of having RFU context can justify the cost of the incoming additional storage and architectural complexities.

Many contemporary high-performance processors support out-of-order execution and speculation. When a context-full RFU is integrated into a speculative processor, if an RFU instruction has modified the RFU context but was mis-speculated and cannot be committed in the end, the RFU context must be recovered as if the instruction had not been executed.

Traditional mechanism to handle speculative executions of instructions cannot effectively address this issue. As the total size of the architecture registers in a processor is fixed and is not very large, we can build a reorder buffer or a physical register file with reasonable size to save the outputs of the speculative executions of the instructions. But in a context-full RFU, the size of the RFU context varies with different computations and the number of speculative updates on the RFU context may be large, so in the worst case the total storage for the RFU context could be unacceptably large for all the speculative updates to be saved inside the RFU before they are committed. So considering the cost, we do not believe it feasible to track each update to the RFU context on a per-RFU-instruction basis.

It is also not feasible either to save the RFU context in the conventional register file at the update on the RFU context, both due to the inadequate space in the register file and due to the communication overhead

between the RFU and the register file. Though the approach of a shadow register [16] partly addresses this issue, it cannot be fully deployed for context-full RFUs.

Because of this complexity, previous approaches which integrated RFUs into processors either did not support context-full RFUs in speculative processors [2, 3, 9], or did not employ speculative processors as the base processors [4], or did neither of them [1].

In this paper, we propose an architecture and mechanism for supporting speculative execution of a context-full RFU in an in-order issue, out-of-order execution processor. It does not require too much extra size for the RFU context storage and the performance penalty is low in practice.

2. RELATED WORK

The most widely used type of RL is its fine-grain representative – FPGA. Some publications proposed coarser-grain RL, such as MATRIX [5] and PipeRench [6], to achieve higher performance in some computations. MATRIX was an array of 8-bit function units with configurable network. PipeRench consisted of many stripes, each of which had 16 8-bit processing elements. These RL systems were not designed to be integrated into processors.

In order to leverage advantages of both processors and RL, many researchers have tried to integrate RL into processors since the 1990s, for example PRISC [1], DISC [7], Garp [4], MorphoSys [8], OneChip [9], Amalgam [10], Chimaera [2, 3], AMBER [11].

PRISC [1] was one of the early attempts of the integration. It integrated an RFU into "a mythical 200MHz MIPS R2000 datapath". The RFU was composed of an array of look-up tables (LUTs) evaluating 32-bit binary functions. PRISC claimed that on average it could accelerate SPECInt92 by 22%. It did not provide context support inside the RFU nor did it have a speculative processor as its main processor.

Garp [4] integrated an RFU into a simple MIPS II processor. The RFU normally composed of 32 rows, each of which could support 32-bit operations. Garp claimed 2-24x speedups for some applications over an UltraSPARC processor. It supported RFU context but did not support speculative execution.

Chimaera [2, 3] integrated an RFU into an aggressive, dynamically-scheduled MIPS superscalar processor. The RFU was a reconfigurable array containing multiple rows. Each row contained multiple LUT-based reconfigurable blocks and supported up to 32-bit bit-level operations. Chimaera resulted in 21% performance improvements for MediaBench and Honeywell [2], speedup of two or more for some general-purpose computations, and a potential speedup of 160 for some hand-mapped applications (e.g. Game of Life) [3]. Though Chimaera employed a speculative processor as its base processor, the authors indicated that due to the complexity of possible speculative updates on the context and some other reasons, they did not support RFU context [3].

Cong et al. proposed an approach of using shadow registers to selectively copy the execution results of an RFU in the write-back stage, which could efficiently reduce the communication overhead between the processor core and the RFU [16]. However, they assumed that the required number of shadow registers was usually much smaller than the register file, which is not the case for a context-full RFU. Another constraint in their approach was that a shadow register should remain at its proper value (without being overwritten by other instructions) during the time when the RFU reads that register and the time when it completed. A context-full RFU does not have such constraint as the RFU context is inside the RFU and cannot be written by other instructions.

3. DESCRIPTION OF THE PROPOSED ARCHITECTURE & MECHANISM

In a processor with a context-full RFU, any RFU instruction could read/write the context, so RFU instructions themselves should be executed in program order. In the proposed architecture and mechanism, snapshots of the execution context are saved periodically inside the RFU and speculatively executed RFU instructions are logged. When an incorrect speculation is detected, the execution context must be rolled back with a correct snapshot and then the logged instructions (except the mis-speculated instruction) can be replayed to update the execution context to a state immediately before the execution of the incorrectly executed instruction.

An elaborate design is required to make this idea correct and efficient. In the implementation, we propose three kinds of context storage to support this idea.

The *Execution Context* is the architecturally-visible context, which is updated by an RFU instruction. We built components to periodically save snapshots of it

for potential roll-backs. One important snapshot is *Committed Context*, where all the RFU instructions that contributed to this context have already been committed. With this snapshot, we can restore the Execution Context to an older but error-free state when necessary. The Committed Context has a special indicator called *Committed Context Owner*, which indicates the most recently committed RFU instruction that has updated the Committed Context.

In order to replay the necessary RFU instructions during the roll-back process, we build a structure called *Inst Queue*. In an in-order issue processor, an RFU instruction is inserted into the Inst Queue in-order when it arrives at the RFU; and it is deleted when it is no longer younger than the Committed Context Owner or when it is directly or indirectly killed. During the roll-back process, the instructions in the Inst Queue will be replayed, and their outputs are ignored except the updates to the RFU context. And in the non-RFU roll-back process, the register file outside the RFU can be restored as usual and the non-RFU instructions can be replayed outside the RFU.

A first thought for generating the data of the Committed Context might be copying the value of the Execution Context when an RFU instruction *X* is committed. But this may not be correct, because there is some latency between the completion of X's execution and its commitment, the RFU instructions younger than X may have updated the Execution Context before X is committed; then when X is committed, the Execution Context may no longer reflect the effect made by X and might be incorrectly updated by some incorrectly speculatively executed RFU instruction younger than X.

The solution is to keep another snapshot of the Execution Context. This snapshot may reflect the effect of a speculatively executed RFU instruction. We call this snapshot *Speculation Context*.

The Speculation Context is updated periodically (after a predefined number *K* RFU instructions are committed; the exact value of K depends on the implementation optimization) with the value of the Execution Context. It also has a special indicator called *Speculation Context Owner*, which indicates the most recent RFU instruction having executed that may reflect the update of the Speculation Context.

When the Speculation Context Owner is committed, the Committed Context can be updated with the value of the Speculation Context. As the Speculation Context Owner is the youngest instruction that may update the Speculation Context and it has already been committed at that time, the Committed Context is guaranteed to have a version of context where all the RFU instructions having updated it have committed.

When the roll-back process is triggered by an incorrect speculation, the Execution Context will be recovered with the value of the Committed Context, and then the logged instructions in the Inst Queue can be replayed to update the Execution Context to a state immediately before the execution of the incorrectly speculatively executed RFU instruction.

Figure 2 shows a high-level diagram on the relationship among those three RFU context components.

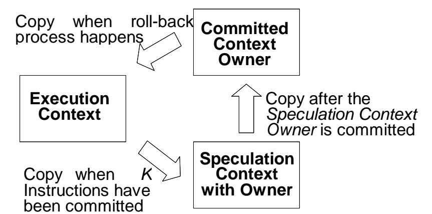

Figure 2: High-level diagram on the relationship among the three contexts.

Note that at any time, the architecturally-visible context is always the Execution Context. The Speculation Context and the Committed Context take effect only in the roll-back process.

Figure 3 shows all the architecture components for supporting speculative execution of RFU instructions. An RFU instruction contains a tag, operation (op), source operands values (src) and return destination (dest) fields. An RFU COMMIT message is sent by the ReOrder Buffer to the RFU when an RFU instruction is committed. It contains the tag of an RFU instruction. The Counter is inside the RFU and is increased by one every time the RFU receives an RFUCOMMIT message. When it reaches a predefined number *K,* the Speculation Context is updated with the value of the Execution Context, and then it will be reset to 0. It will also be reset to 0 when an RFU instruction is killed.

There might be three kinds of penalties introduced by the proposed architecture & mechanism. The first one is the storage penalty. It is mainly from the cost for the Speculation Context and the Committed Context, so the total additional space requirement is two times that of the Execution Context, which is within a reasonable scope.

The second kind of penalty is the performance penalty by the additional logic for handling the

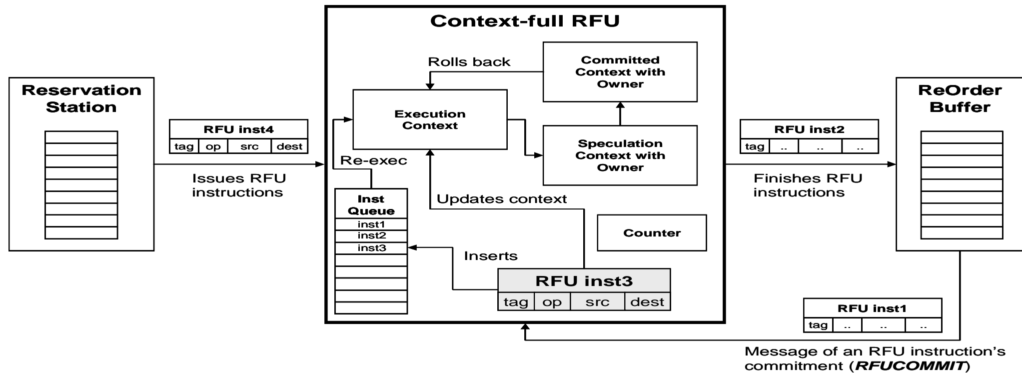

Figure 3: Architecture components for supporting speculative execution of RFU instructions.

copying of the contexts. Since this logic is within the RFU and is invisible to the components outside the RFU, we can hide the latency of this logic, so this kind of penalty should be negligible.

The third kind of penalty is the performance penalty which comes from the roll-back process. Since in contemporary speculative processors the speculation error rate is very low, for example normally less than 5% in branch predictions [12], this kind of penalty is effectively low in practice.

4. EVALUATION

In order to get a credible result in the performance evaluation, we employed a cycle-accurate simulation methodology to evaluate workloads on the proposed architecture and mechanism.

ASIM is a cycle-accurate system-level simulation framework/system [13]. In ASIM, we used a processor core module reflecting the most recent Intel processor Core i7 with a slight difference. The frequency was 2.4 GHz. The system memory size was 16GB, and the operating system was Linux 2.6.5. For simulating the RFU, we wrote an RFU sub-module into the processor core module. The RFU was located between the reservation station and the reorder buffer and it acted like a normal function unit. It worked in a fixed frequency which would divide exactly into the frequency of the processor.

The performance was measured in CPU cycles. New RFU instructions were added and we normalized the RFU cycles to the CPU cycles considering the difference between the CPU frequency and the RFU frequency. We modified the C source code for the workloads into C codes with embedded assembly language codes using RFU instructions. In the simulation, an RFU instruction was dispatched into the RFU for execution. In the experiment, the value of *K* was set to 5.

We had three sets of data in the evaluation: 1) the data for the workloads on an architecture without handling any incorrect RFU context update by the speculative execution of the RFU instructions (labeled as *SPEC_INCO*). In this case, the workloads might have incorrect result; 2) the data for the workloads on architecture without the speculative execution of the RFU instructions (labeled as *NONE_SPEC*). In this case, the workloads would have correct result but would run slower; 3) the data for the workloads on the proposed architecture and mechanism (labeled as *PROPOSED*). In this case, the workloads would generate correct results and we will see that the performance penalty is negligible, especially when the speculation error rate is low.

The first workload was a toy program. It conditionally incremented the RFU context by 1 using an RFU instruction. In the evaluation, we adjusted the speculation error rate to be: 0%, 1%, 3%, 5%, 7% and 10%. The iteration count was set to 10,000.

In table 1, we show the RFU roll-back times in PROPOSED, the count of replayed RFU instructions in PROPOSED, the execution cycle numbers on different architectures, and the speedup of PROPOSED over NONE_SPEC. Note that with different speculation error rates, the RFU instructions were executed different numbers of times. In SPEC_INCO the program may execute incorrectly with a wrong instruction flow, so the execution cycle numbers in PROPOSED were sometimes less than those in SPEC_INCO while at other times they were greater, and sometimes they happened to be equal. From the performance point of view, we can see that PROPOSED got significant speedups over NONE_SPEC.

Table 1. The evaluation data for the toy program

Speculation error rate		0%	1%	3%	5%	7%	10%
RFU roll-back times in PROPOSED		0	106	321	494	683	1038
Replayed RFU instructions count in PROPOSED		0	204	659	1004	1311	2045
Execution cycle number for the evaluated codes	**NONE_SPEC**	1329920	1323378	1311253	1300505	1290137	1270778
	PROPOSED	999918	1002962	1010540	1015220	1022463	1036141
	SPEC_INCO	999918	1002962	1010550	1015220	1022463	1036131
Speedup of PROPOSED over NONE_SPEC		1.33	1.32	1.30	1.28	1.26	1.23

The second workload in the evaluation was the kernel part of a real application: Perl Compatible Regular Expression (*PCRE*), which is a regular expression C library [14]. We implemented an RFU version of PCRE, in which a part of a pattern (pattern unit) is fed into the RFU, and then the RFU performs the matching by comparing the pattern against the text. This was done by an RFU instruction *RFU_pcre*. Because a pattern is used to match a very long text, the RFU context can be employed to store the current pattern unit to avoid repeatedly sending it into the RFU. An RFU instruction *RFU_putcontext* is used to put various pattern units into RFU context. After the execution of one RFU_pcre instance, the RFU might either require a new pattern unit or not; the executions of RFU_putcontext are determined by the results of RFU_pcre. Then in a speculative processor, RFU_putcontext can be speculatively executed and speculatively update the RFU context with new pattern units.

Table 2 shows the evaluation data for PCRE. The data set used was the LLDOS 1.0 - Scenario One in dataset 2000 of the DARPA Instruction Detection Data Sets [15], with 120MB network data in total. We tried to compare all the data against all the 2223 distinct regular expression patterns that appeared in the 9108 regular expression rules in Snort of the 2008-04-22 version. And there were only 38 patterns that were compared against in practice. We could not get the SPEC_INCO result for eight patterns as the program went nowhere with incorrect speculation for these patterns. Because of the space limitation, in table 2 we only show the data for three patterns with the highest speedups, three patterns with the lowest speedups, and the aggregate results for all the 30 patterns where we could get the SPEC_INCO results.

There were many patterns to be matched. In some cases there were many dependencies between the condition instructions and the RFU instructions but in some other cases there were not (where the proposed mechanism had few chance to show its advantages). So we can see that the speedups of PROPOSED in different patterns varied. Overall, PROPOSED still got satisfying speedups in this complex workload.

Table 2. The evaluation data for PCRE

Patterns		**With the highest speedup**			**With the lowest speedup**			**Aggregate**
		#4	**#5**	**#16**	**#25**	**#37**	**#38**	
RFU roll-back times in PROPOSED		80	67	13	1088	0	0	61519
Replayed RFU instructions count in PROPOSED		145	100	11	544	0	0	102075
Execution cycle number for the evaluated codes	**NONE_SPEC**	128513	62458	9632	2221667	840655	840655	102651567
	PROPOSED	115119	56970	8567	2220477	839930	839930	97837327
	SPEC_INCO	115119	56970	8567	2220477	839930	839930	97358394
Speedup of PROPOSED over NONE_SPEC		1.12	1.10	1.12	1.00	1.00	1.00	1.05

From the evaluation, we can see in all the evaluation data the execution cycle number of PROPOSED was very close to SPEC_INCO. Since there was no performance penalty in SPEC_INCO (though the result might be incorrect), we can infer that the performance penalty in the proposed architecture & mechanism was low in practice.

5. CONCLUSION

RL can provide much better performance than conventional processors for some workloads and has much more flexibility than fix-function logic. Recently researchers have interest in integrating RL into processors as RFUs. A context-full RFU can eliminate some unnecessary data movement overheads and has some other benefits. However, due

to the design complexity, previous approaches did not support context-full RFUs in speculative processors.

We proposed an architecture and mechanism for supporting speculative execution of a context-full RFU. With an elaborate design, it does not require too much extra size for the RFU context storage. The evaluation data showed that the performance penalty would be low in practice.

REFERENCES

[1] R. Razdan and M.D. Smith. "A high-performance microarchitecture with hardware-programmable functional units". in *Proc. 27th International Symposium on Microarchitecture*, Nov. 1994, pp. 172-180.

[2] Z-A. Ye, A. Moshovos, S. Hauck and P. Banerjee. "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit". in *Proc. the 27th International Symposium on Computer Architecture*, June 2000, pp.225-235.

[3] S. Hauck, T.W. Fry, M.M. Hosler and J.P. Kao. "The Chimaera reconfigurable function unit". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 12, No. 2 (2004), pp. 206-217.

[4] J.R. Hauser and J. Wawrzynek. "Garp: a MIPS processor with a reconfigurable coprocessor". in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 12-21.

[5] E. Mirsky, A. DeHon. "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources". in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 157-166.

[6] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine and R.R. Taylor. "PipeRench: a virtualized programmable datapath in 0.18 Micron Technology". in *Proc. 2002 IEEE Custom Integrated Circuits Conference*, May 2002, pp. 63-66.

[7] M.J. Wirthlin and B.L. Hutchings. "A dynamic instruction set computer". in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, April, 1995, pp. 99-107.

[8] G. Lu, H. Singh, M-h. Lee, N. Bagherzadeh, and F. Kurdahi. "The MorphoSys Parallel Reconfigurable System". in *Proc. the 5th International Euro-Par Conference on Parallel Processing*, Aug.-Sept. 1999, pp. 727-734.

[9] J.E. Carrillo and P. Chow. "The effect of reconfigurable units in superscalar processors". in *Proc. International Symposium on Field Programmable Gate Arrays*, Feb. 2001, pp. 141-150.

[10] D.B. Gottlieb, J.J. Cook, J.D. Walstrom, S. Ferrera, C-W. Wang and N.P. Carter. "Clustered programmable-reconfigurable processors". in *Proc. 2002 IEEE International Conference on Field-Programmable Technology*, Dec. 2002, pp. 134-141.

[11] F. Mehdipour, H. Noori, M.S. Zamani, K. Murakami, M. Sedighi and K. Inoue. "An integrated temporal partitioning and mapping framework for handling custom instructions on a reconfigurable functional unit". *ACSAC 2006, LNCS 4186*, pp. 219-230.

[12] Harish Patil and Joel S. Emer. "Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing". *Sixth International Symposium on High-Performance Computer Architecture*, 2000, pp. 251-263

[13] J. Emer, P. Ahuja, E. Borch, A. Klauser, C-K. Luk, S. Manne, S.S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: a performance model framework", *IEEE Computer*, Vol 35, Issue 2 (Feb 2002), pp. 68-76.

[14] PCRE - Perl Compatible Regular Expressions, http://www.pcre.org/

[15] Lincoln Laboratory Scenario (DDoS) 1.0. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/2000/LLS_DDOS_1.0.html

[16] Jason Cong, Yiping Fan, Guoling Han, Ashok Jagannathan, Glenn Reinman, Zhiru Zhang, "Instruction Set Extension for Configurable Professors with Shadow Registers", *FPGA'05*(February 20–22, 2005, Monterey, California, USA). pp. 99-106