TOWARDS A VIABLE OUT-OF-ORDER SOFT CORE: COPY-FREE, CHECKPOINTED REGISTER RENAMING

Kaveh Aasaraai and Andreas Moshovos

Electrical and Computer Engineering
University of Toronto
{aasaraai, moshovos}@eecg.toronto.edu

ABSTRACT

As a step torward a viable, single-issue out-of-order soft core, this work presents *Copy-Free Checkpointing* (CFC), an FPGA-friendly register renaming design. CFC supports speculative execution by implementing checkpoint recovery. Compared against the best conventional register renaming implementation CFC requires 7.5x to 6.4x fewer LUTs and is at least 10% faster.

1. INTRODUCTION

Embedded systems increasingly exploit the cost effectiveness and flexibility of FPGAs. Such FPGA-based systems often include soft processors for several reasons; for example, certain tasks are best, cost- or performance-wise, implemented in processors, whereas processor-based implementations can be faster and easier to develop, and debug than custom accelerators. If history is any indication of the future of embedded systems, their functionality will increase and their applications will evolve increasing in complexity, footprint, and functionality (cell phone designs, for example, have followed such trends). Accordingly, it is important to develop higher performing embedded processors.

While hard cores, either embedded in or external to the FPGA, could offer high performance, both options have their shortcomings: Embedded hard cores are wasted when not needed and are inflexible while external hard cores increase cost and suffer from inter-chip communication latencies. Soft cores are a suitable alternative as long as they can offer sufficient performance.

Besides special purpose accelerators (such as those implementing specialized instructions), techniques that improve performance generally rely on increasing the concurrency of instruction processing. Such techniques include pipelining, Superscalar [1], Very Long Instruction Word (VLIW) [1], Single Instruction Multiple Data (SIMD), and Vector [1, 2] execution. VLIW, SIMD and Vector execution exploit programmer- or compiler-extracted instruction-level parallelism. When this is possible, each of these alternatives has specific advantages.

However, there are applications where parallelism is less structured and difficult to extract. It is such parallelism that superscalar and out-of-order (OOO) execution target [1]. Superscalar processors use multiple datapaths to *completely* overlap the execution of multiple adjacent instructions. OOO processors allow instructions to execute in any order that does not violate program semantics and thus OOO can extract more parallelism than superscalar execution [1, 3]. OOO execution can be used with a single or multiple datapaths and can extract even more parallelism via register renaming and speculative execution (Section 3).

While many general-purpose, hard cores use OOO execution, soft cores do not. The main reason is that OOO structures have been developed for custom logic implementations and are not necessarily well-suited for an FPGA substrate. However, by revisiting OOO structure design it may be possible to rip most of the benefits of OOO execution even on an FPGA substrate. Accordingly, the goal of this work is to develop such *FPGA-friendly* OOO soft core designs.

This work demonstrates that single-datapath, or *single-issue* OOO execution has the potential of improving performance compared to superscalar execution. This potential will materialize only if it will be possible to develop cost-effective and fast FPGA-friendly OOO cores. As a first step towards this goal, this work presents *Copy-Free-Checkpointed Renaming* (CFC), an FPGA-friendly register renaming design. CFC can rename a single instruction per cycle, and supports speculative execution.

Using the the Altera NIOS II instruction set architecture this work compares CFC against implementations of the conventional SRAM-based (RAT_{RAM}) [4] and CAM-based (RAT_{CAM}) [5] register renaming designs. RAT_{RAM} and RAT_{CAM} require specialized functionality which requires a large number of LUTs in an FPGA. CFC is designed to use block RAMs (BRAMs) for most of its state, resulting in a much lower and more balanced utilization of FPGA resources. CFC can be pipelined and operates at a higher frequency than RAT_{RAM} and RAT_{CAM} and thus performs better. While this work focuses on checkpointed register renaming, the concept of CFC can be applied in

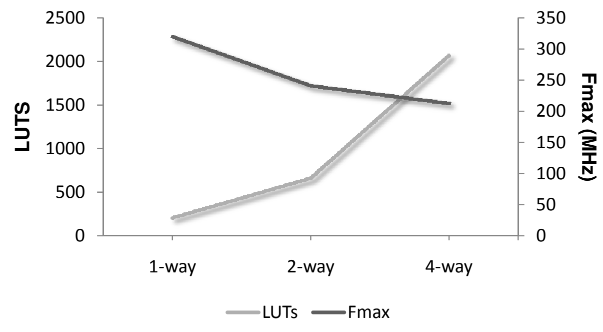

Fig. 1. Area and maximum frequency of 1-, 2-, and 4-way superscalar datapaths on a Stratix III FPGA.

other applications where checkpointing is needed. FPGA-friendly implementations of the other key OOO processor components such as the scheduler is left for future work.

2. MOTIVATION

This section demonstrates that single-issue OOO execution has the potential of being faster than superscalar execution. Both architectures target applications with unstructured, hard-to-find instruction-level parallelism.

An *N-way* superscalar can execute up to *N* adjacent in the original program order instructions in parallel. To do so most of the datapath components are replicated *N* times. Bypass paths are needed among these *N* datapaths to avoid unnecessary stalls. Accordingly, supercalar resource costs increase super-linearly with the number of ways. Figure 1 shows how the area and frequency of a superscalar datapath scale as the number of ways increases from one to four. For this experiment each datapath uses a classical, five-stage pipeline with full bypass paths, a 32-bit adder, and the pipeline latches. No other components are modeled. The maximum frequency of the 4-way superscalar is 33% less than that of the single-issue processor. The 4-way superscalar must extract sufficient instruction level parallelism (ILP) to overcome this frequency disadvantage.

OOO processors may execute an instruction as long as this does not violate any program dependencies [1, 3]. OOO can extract more ILP than a superscalar since in OOO, instructions executing in parallel do not have to be adjacent. A further boost in ILP comes from *register renaming* that eliminates all but true (read-after-write) data dependencies. General-purpose, hard processors combine OOO and superscalar execution since resources are plentiful (nearly a billion of transistors is common today). When maintaining a low resource usage is important, as it is in an FPGA substrate, OOO can be used with just one way. In this case, OOO improves performance over simple pipelining by not stalling when an instruction requires additional cycles to execute. Waiting for memory is, for example, a major source of delays even for soft cores.

Figure 2 compares the "instructions per cycle" (IPC) performance of 1-, 2-, and 4-way superscalar, and single-issue OOO processors, for a wide range of cache configurations (Section 6.1 describes the methodology). Cache size varies from 4KB up to 32KB (stacked bars), while cache associativity varies from one (part (a)) to two (part (b)). Split instruction and data caches of the given capacity are modeled. OOO outperforms both 1- and 2-way superscalars for all cache sizes, while it performs worse than the 4-way superscalar only with the 32KB cache. A better compiler could improve performance for the superscalar processors.

The results of this section demonstrate that OOO has the potential to improve performance on top of simple pipelining while avoiding the super linear costs of datapath replication of superscalar execution. Several challenges remain for this potential to materialize. First, performance depends also on the operating frequency. Second, OOO introduces additional structures on top of a 1-way processor. The resulting cost and performance must be better than that of 2- or 4-way superscalars for OOO to be a better alternative.

Ideally, existing OOO designs would map easily onto FPGAs achieving reasonable performance and resource usage. However, the FPGA substrate is different than that of ASICs and exhibits different trade offs. Accordingly, it is necessary to revisit OOO design while taking the unique characteristics of FPGAs into consideration.

3. REGISTER RENAMING

Register renaming eliminates some of the dependencies that OOO execution must preserve exposing more ILP. There are three dependency types that must be preserved: read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). WAR and WAW are *false dependencies* as they are an artifact of re-using a limited number of registers. Register renaming eliminates false dependencies by mapping, at run time, the architectural registers referred to by instructions to a larger set of physical registers implemented in hardware. False dependencies are eliminated by using a different physical register for each write to the same architectural register.

At the core of register renaming, is a register alias table (RAT) which maps architectural (RAT index) to physical (RAT values) registers [1, 3, 4, 5]. Renaming an instruction for a three operand instruction set such as that of Nios II proceeds as follows: First, the current mapping of the two source registers are read from the RAT. Then, a new mapping is created for the destination register. A *free list* provides the new physical register name. The processor recycles a physical register when it is certain that no instruction will ever access its value (e.g., when a subsequent instruc-

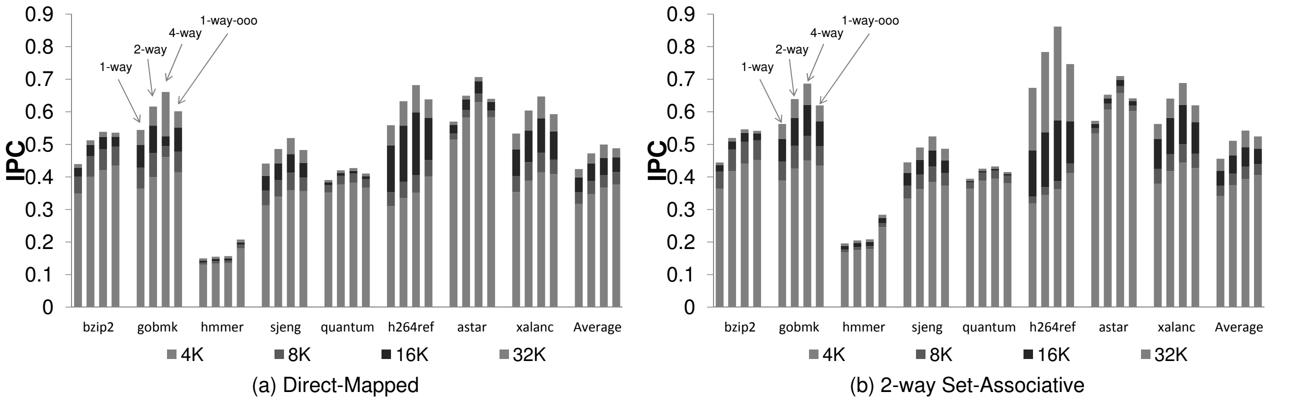

Fig. 2. IPC performance of superscalar and out-of-order processors as a function of cache size and associativity: (a) direct mapped, and (b) 2-way set associative.

tion that overwrites the same architectural register commits).

3.1. The Need for Checkpointing

OOO processors also use speculative execution to boost performance where they execute instructions without being certain that they should. A common form of speculative execution is based on control flow prediction where the processor executes instructions starting at the predicted target address of a branch. When the speculation is correct, performance may improve because the processor had a chance of executing instructions earlier than it would if it had to wait for the branch to decide its target. When the speculation fails, all changes done, including any RAT updates, by the erroneously executed instructions must be undone. For this purpose OOO processors rely on the *re-order buffer* (ROB) [1]. The ROB records, in order, all the RAT changes done by instructions as they are renamed. To recover from a misspeculation, the processor processes the ROB in reverse order, reverting all erroneous RAT updates.

Recovery via the ROB is slow, and requires time that is proportional to the number of erroneously executed instructions. For this reason, many OOO processors include *checkpoints*, a recovery mechanism that has a fixed latency, often a single cycle. For the RAT, a checkpoint is a complete snapshot of its contents. Checkpoints are expensive to build, and increase RAT latency [4, 6, 7]. Accordingly, only a few of them are typically implemented.

When both checkpoints and an ROB are available, recovery can proceed as follows: If the mis-speculated instruction has a checkpoint, recovery proceeds using that checkpoint alone. Otherwise, recovery proceeds at the closest *subsequent* checkpoint first, and then via the ROB to the relevant instruction [7]. Alternatively, the processor can recover to the closest *preceding* checkpoint at the expense of re-executing any intervening instructions [6]. In this case

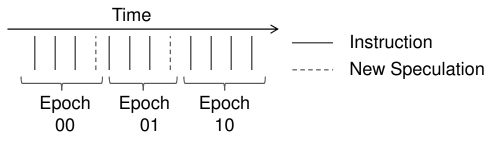

Fig. 3. Epochs illustrated in a sequence of instructions.

an ROB is unnecessary. A few checkpoints offer performance that is close to that possible with an infinite number of checkpoints [6, 7]. Accordingly, we limit our attention to four or eight checkpoints.

RAT_{RAM}, a common RAT implementation, is a main table indexed by architectural register names and whose entries contain physical register names [4]. For the NIOS II instruction set, this table needs three read ports, two for the source operands plus one for reading the previous mapping for the destination operand to store it in the ROB. The main table also needs one write port to write the new mapping for the destination register. A checkpoint is a snapshot of the table's content and is stored in a separate checkpoint table. Multiple checkpoints require multiple checkpoint tables. Recovery amounts to copying back a checkpoint into the main table. A checkpoint is taken when the processor renames an instruction which initiates a new speculation (e.g., a branch). Such an instruction terminates an *epoch* comprising the instructions seen since the last preceding checkpoint as shown in Figure 3. Recovering at a checkpoint effectively discards all the instructions of all subsequent epochs.

4. COPY-FREE CHECKPOINTING (CFC)

CFC modifies RAT_{RAM} to better match an FPGA substrate. The key challenge when implementing RAT_{RAM} on an FPGA is the implementation of the checkpoints. Creating checkpoints requires copying all bits of the main table into one of the checkpoint tables. In a custom logic implementation, the checkpoints are implemented as small queues that are embedded next to each RAT bit. This implementation is expensive and inefficient on an FPGA since it uses LUTs exclusively.

In RAT_{RAM}, the main table holds all the changes applied to the RAT by all the instructions, both speculative and non-speculative. The advantage of this implementation is that the most recent mapping for a register always appears at the corresponding entry of the main table. Hence lookups are streamlined but checkpoints need to take a complete snapshot of the main table.

Instead of storing updates always on the same main table, CFC uses a set of BRAM-implemented tables and manages them as a circular queue. CFC allocates a separate table for each epoch and all updates from within this epoch are done to the epoch's table. Therefore, recovering from a mis-speculated epoch is as simple as discarding the corresponding table and all tables that follow. While this streamlines RAT checkpointing, RAT lookups are not as straightforward as in RAT_{RAM}. To find the most recent mapping of a register, the processor has to search through the active tables in order, starting form the most recent one. Fortunately, as explained in the rest of this section, implementing these ordered searches is relatively inexpensive using a few LUTs. A separate table contains the register mappings done by instructions as they commit. This is necessary to recover from unexpected events such as page faults or interrupts.

4.1. The New RAT Structure

Figure 4 shows that CFC comprises two main structures: the RAT tables and the dirty flag array (DFA). Each RAT table has one entry per architectural register, each containing a physical register name. There are C+1 tables, C for the checkpoints and one for the committed table. CFC uses two pointer registers, head and tail, to manage the tables in a circular queue. The DFA is organized as a grid of bits with one row per architectural register, and one column per RAT table and tracks the valid mappings in each checkpoint table. When a new table is allocated, all corresponding DFA bits are cleared indicating that the table contains no valid mappings. As instructions are renamed, they set the corresponding DFA bits. The (C+1)th table, the *committed table*, represents the RAT's architectural state, being the latest changes applied by non-speculative instructions.

4.2. RAT Operations

This section explains how CFC performs the various renaming operations. Initially, only the committed table is active containing a valid mapping of architectural to physical registers. Since an instruction is speculative before it is committed, execution commences by first allocating a new epoch

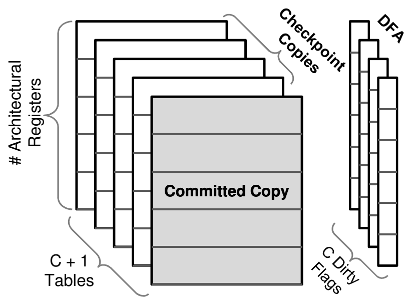

Fig. 4. CFC main structure consists of C+1 tables and a dirty flag array.

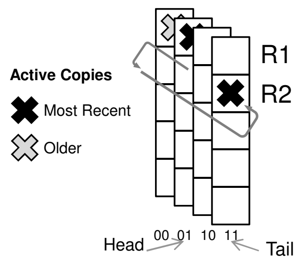

Fig. 5. Finding the most recent mapping: The most recent mapping for register R1 is in the second table, while for R2, is in the fourth.

and by initiating a checkpoint for it.

Initiating a Checkpoint: CFC initiates a checkpoint by (1) advancing the head pointer, and (2) clearing all DFA bits on the corresponding column. As CFC directs all subsequent RAT updates to this new table, any preceding tables remain intact, and can be used for recovery. No copying is necessary when initiating a new checkpoint.

Finding the Most Recent Mapping: To rename a source operand, the processor needs to find the table holding the most recent mapping. This is achieved by searching through the DFA row of the architectural register. Conceptually, this search is done sequentially, looking for the first set dirty flag, starting from the head and moving backward toward the tail. If no dirty flag is found set, then the committed table is used. Figure 5 shows two examples. On an FPGA, this search can be easily implemented as a lookup table having as inputs the DFA row and the two pointers.

Creating a New Mapping: To rename a destination register, CFC stores a new mapping into the active table (pointed to by the head pointer), and sets the corresponding DFA bit.

Committing a Checkpoint: When an instruction commits, CFC writes its destination register mapping into the committed table. As a result, the committed table always contains a valid RAT state that the processor can use to recover from any exceptional event. CFC recycles checkpoints by advancing the tail pointer upon commit of an instruction that allocated a checkpoint.

Restoring from a Checkpoint: On a mis-speculation the RAT must be restored to the state it had before renaming any of the mis-speculated instructions. Simply reverting the head pointer to the epoch of the mis-speculated instruction is sufficient. All subsequent tables are effectively discarded. Restoring from a checkpoint does no copying either.

5. MAPPING ONTO AN FPGA

This section details how CFC is implemented on an FPGA. Most of the RAT state is stored in BRAMs. BRAMs are high-speed, area-efficient memory arrays that significantly increase design efficiency.

Flattenning: The DFA logic determines which table T contains the most recent mapping. When reading from the checkpoint tables, a C-to-1 multiplexer selects the most recent mapping if any. This multiplexer is area and latency inefficient. Instead of using separate BRAMs for each of the C tables, and then selecting among those, the tables can be combined into a single table, eliminating the multiplexer. The entry for the architectural register A of table T now appears at row "$A \times C + T$" of the flattened table. As long as C is a power of two, the flattened table can be accessed easily by concatenating A and T. Alternatively, the index can be "$T \times 32 + A$", in which case, indexing through concatenation of A and T is still possible as long as the number of architectural registers A, is a power of two.

Multiporting the RAT: Two processor stages access the CFC tables: rename and commit. Renaming an instruction requires reading at most three, and writing one mapping. Committing an instruction requires writing a mapping into the committed copy. In total, the tables must have three read and two write ports. Unfortunately, BRAMs have only one read and one write port. Fortunately, the commit stage writes only to the committed copy, while the rename stage writes only to the checkpoint copies. Using separate BRAMs, one for the committed copy and the other for the flattened table allows to use single write-ported BRAMs. For RAT lookups, a 2-to-1 multiplexer selects between the two BRAMs. This multiplexer does not add significant area or latency overhead. Replicating each of the two BRAMs three times provides the three read ports needed during the rename stage. Writes are performed to all three copies simultaneously to keep them coherent.

Dirty Flag Array: The DFA is accessed one row at a time, but is reset one column at a time. Therefore, DFA and the associated logic are best implemented using LUTs.

Pipelining the CFC: Compared to RAT_{RAM}, CFC adds a level of indirection prior to accessing the tables; a DFA access determines where the most recent copy is. Consequently, CFC can be slower than RAT_{RAM}. Fortunately, CFC can be pipelined into two stages as follows: In the first stage, CFC accesses the DFA, generates a BRAM index for the flattened table, and updates the DFA. In stage two, CFC accesses the checkpoint and committed BRAMS in parallel, and at the end, selects the appropriate copy. At the end of the second stage, all BRAM updates occur as well.

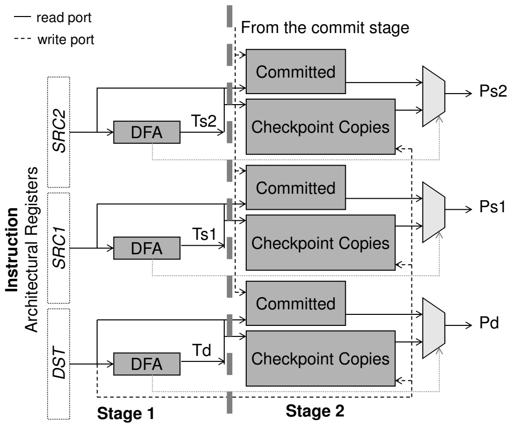

Fig. 6. CFC Implementation.

Figure 6 shows the overall architecture of the CFC implementation where the architectural registers SRC1, SRC2 and DST of an instruction are renamed into the physical registers Ps1, Ps2, and Pd respectively.

6. EVALUATION

This section compares the performance and cost of CFC, RAT_{RAM}, and RAT_{CAM}. Section 6.1 details the experimental methodology. Section 6.2 reports the LUT usage of each method, while Section 6.3 reports their operating frequencies. Section 6.4 measures the impact of CFC pipelining on IPC performance. Finally, Section 6.5 reports overall performance and summarizes our findings taking into account LUT and BRAM usage.

6.1. Methodology

We implemented the three renaming schemes using Verilog. We used Quartus II v8.1 for synthesis and place-and-route for the Altera Cyclone II (mid-range – 35K LUTs) and Stratix III (high-end – 150K LUTs) FPGAs.

To estimate execution performance, and given that we do not have a full OOO Verilog implementation yet, we developed a cycle-accurate Nios II full system simulator capable of booting and running the uCLinux operating system [8]. Table 1 details the simulated superscalar and out-of-order processor architectures. We use benchmarks from the SPEC CPU 2006 suite that are typically used to evaluate the performance of desktop systems [9]. We use them as representative of applications that have unstructured ILP assuming that in the future, embedded or FPGA-based systems will be

Table 1. Simulated processors.

Common Properties		**Superscalar Specific**	
BPredictor Type	Bimodal	Pipeline Stages	5
Bimodal Entries	512	**Out-of-Order Specific**	
BTB Entries	512	Pipeline Stages	7
Cache Size (Bytes)	4K, 8K, 16K, 32K	Scheduler Size	32
Cache Associativity	Direct, 2-way	ROB Size	32
Memory Latency	20 Cycles	Physical Registers	64
		Checkpoints	4

Table 2. LUT usage and maximum frequency for three renaming methods with four or eight checkpoints on different platforms.

	Platform	**Chkpnts**	**CFC**	**RAT$_{CAM}$**	**RAT$_{RAM}$**
LUTs	Cyclone II	4	501	2378	3220
		8	964	3631	6368
	Stratix III	4	399	1802	3002
		8	996	2327	7082
Freq.	Cyclone II	4	137	85	122
		8	104	71	82
	Stratix III	4	292	133	195
		8	220	105	196

called upon to run demanding applications such as these (an assumption motivated by past experience). We used a set of reference inputs for running the benchmarks. Input data files are stored in main memory and are accessed through the ramdisk driver. Since Nios II does not have a floating point unit, only a subset of the SPEC suite could be simulated. Measurements are taken for sample of one billion instructions, after skipping several billions of instructions so that most of the execution is past initialization. We consider designs with four or eight checkpoints as past work has shown that they are sufficient [6, 7]. Given the frequencies achieved and the latency of modern memory devices, memory latency is estimated at 20 cycles.

6.2. LUT Usage

Table 2 reports the number of LUTs used by the three renaming methods with four and eight checkpoints on the two FPGA platforms. Since only the DFA and its associated logic uses LUTs in CFC, whereas RAT$_{RAM}$ and RAT$_{CAM}$ use LUTs exclusively, CFC's LUT usage is considerably lower. For example, with eight checkpoints on Stratix III, CFC uses approximately 2x and 7x less LUTs than RAT$_{CAM}$ and RAT$_{RAM}$ respectively. On Cyclone II and with eight checkpoints, CFC uses 2.73% of the available LUTs, while RAT$_{CAM}$ and RAT$_{RAM}$ use 10.37% and 18.19% respectively. CFC uses six BRAMs also, which are only a small fraction of the BRAMs available on either platform.

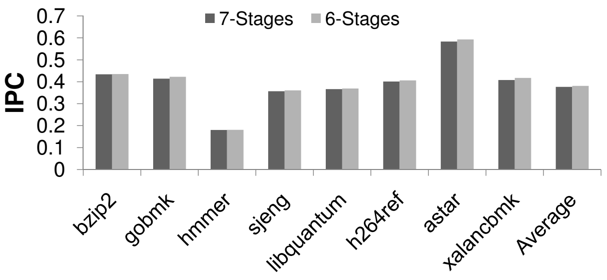

Fig. 7. Performance impact of an extra renaming stage.

6.3. Frequency

Table 2 reports the maximum clock frequency for the three mechanisms. CFC outperforms both conventional schemes on both FPGA platforms. CFC can operate at up to 118% and 50% faster than RAT$_{CAM}$ and RAT$_{RAM}$ respectively.

6.4. Impact of Pipelining on IPC

CFC outperforms RAT$_{RAM}$ and RAT$_{CAM}$, in terms of clock frequency, however, CFC uses two pipeline stages. Pipelining can hurt IPC and thus performance. Figure 7 compares the IPC of single-cycle and two-cycle pipelined renaming (the base architecture has a total of six stages, including one for renaming). The performance penalty of the additional renaming stage is negligible. Coupled with the frequency advantage of CFC, we thus expect that CFC will outperform both RAT$_{RAM}$ and RAT$_{CAM}$.

6.5. Performance and Resource Usage

Tables 3 and 4 compare CFC, RAT$_{CAM}$ and RAT$_{RAM}$ with eight checkpoints on the Cyclone II and the Stratix III FPGAs respectively. Four checkpoint results are omitted due to space limitations. The tables report LUT usage, latency (the sum of the latency of both stages for CFC), effective maximum frequency, average IPC, and average runtime in seconds. Focusing on the Stratix III results, RAT$_{RAM}$ and RAT$_{CAM}$ use 4.7% and 1.6% of the available LUTs, while CFC utilizes just 0.7% of the LUTs. CFC uses 4% of the on-chip BRAMs instead. Latency-wise, CFC is 28% faster than RAT$_{CAM}$ but 45% slower than RAT$_{RAM}$. However, because of pipelining, CFC operates at a higher frequency than either RAT$_{RAM}$ and RAT$_{CAM}$. CFC outperforms RAT$_{RAM}$ and RAT$_{CAM}$ by 10% and 51% respectively. For four checkpoints (not shown), CFC uses 7.5x fewer LUTs than RAT$_{RAM}$.

Table 3. Comparison: Eight checkpoints, Cyclone II.

	$\mathbf{RAT_{RAM}}$	$\mathbf{RAT_{CAM}}$	**CFC**
LUTs (count/%)	6368/18.2%	3631/10.4%	964/2.7%
BRAMs (count/%)	0/0%	0/0%	6/5.7%
Latency (ns)	12.11	13.92	13.97
F.Max (MHz)	82	71	104
Avg. IPC	0.38	0.38	0.37
Total Runtime (s)	33.86	39.11	26.99

Table 4. Comparison: Eight checkpoints, Stratix III.

	$\mathbf{RAT_{RAM}}$	$\mathbf{RAT_{CAM}}$	**CFC**
LUTs (count/%)	7082/4.7%	2327/1.6%	996/0.7%
BRAMs (count/%)	0/0%	0/0%	6/4%
Latency (ns)	5.09	9.45	7.38
F.Max (MHz)	196	105	220
Avg. IPC	0.38	0.38	0.37
Total Runtime(s)	14.16	26.44	12.76

7. RELATED WORK

Mesa-Martinez et al. propose implementing an OOO soft core, SCOORE, on FPGAs [10]. They explain why, OOO implementations do not map well on FPGAs and propose several, general remedies. The primary goal of the SCOORE project is that of simulation acceleration. Fytraki and Pnevmatikatos implement parts of an OOO processor on an FPGA for the purpose of accelerating processor simulation as well [11]. Our work is motivated, in part, by the same inefficiencies that these two prior works identified. However, our goal is different. We aim to develop cost- and performance-effective, FPGA-friendly OOO cores for use in embedded system applications.

8. CONCLUSION

This work demonstrated that for applications that have unstructured, hard-to-find instruction-level parallelism, single-issue out-of-order processors have the potential of outperforming two-way superscalar cores offering performance that is close to four-way superscalar processors. For this potential to materialize, it is necessary to develop FPGA-aware implementations of the various units that out-of-order execution requires. This work presented a novel register renaming scheme that results in a balanced use of FPGA resources. The proposed copy-free checkpointed register renaming was shown to be much more resource efficient than conventional alternatives. It was also shown that it can be pipelined offering superior performance. Compared against the best conventional register renaming implementation CFC requires 7.5x to 6.4x fewer LUTs and is at least 10% faster.

Renaming is just one of the units OOO execution requires. Future work can determine whether FPGA-friendly implementations of these units is possible.

While this work focused on register renaming, checkpointing has many other applications. For example, checkpointing can be used in support of alternative execution models such as transactional memory. We expect that the proposed copy-free-checkpointing can be used for such applications as well.

ACKNOWLEDGEMENTS: We would like to thank the revhiewers for their insightful comments and Elias Ferzli, Alireza Heidar-Barghi, Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Greg Steffan, and Jason Zebchuk for reading earlier drafts of this manuscript. This work was supported by an NSERC Discovery grant and equipment donations from Altera. Kaveh Aasaraai was supported in part by an OGSST scholarship.

9. REFERENCES

[1] J. E. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, 1995.

[2] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 61–70.

[3] A. Moshovos and G. S. Sohi, "Micro-Architectural Innovations: Boosting Processor Performance Beyond Technology Scaling," *Proceedings of the IEEE*, vol. 89, no. 11, Nov. 2001.

[4] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[5] T. N. Buti et al., "Organization and Implementation of the Register-Renaming Mapper for Out-of-Order IBM POWER4 Processors," *IBM Journal of Research and Development, Vol. 49, No. 1*, 2005.

[6] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th International Symposium on Microarchitecture*, 2003, pp. 423–434.

[7] A. Moshovos, "Checkpointing Alternatives for High Performance, Power-Aware Processors," in *Proceedings of the 2003 international symposium on Low power electronics and design*, 2003, pp. 318–321.

[8] "Arcturus Networks Inc., uClinux," http://www.uclinux.org/.

[9] Standard Performance Evaluation Corporation, "SPEC CPU 2006," http://www.spec.org/cpu2006/.

[10] F. J. Mesa-Martinez et al., "SCOORE Santa Cruz Out-of-Order RISC Engine, FPGA Design Issues," in *Workshop on Architectural Research Prototyping (WARP), held in conjunction with ISCA-33*, 2006, pp. 61–70.

[11] S. Fytraki and D. Pnevmatikatos, "RESIM: A trace-driven, reconfigurable ILP processor simulator," in *Design and Automation Europe*, 2008.