NOREBA: A Compiler-Informed Non-speculative Out-of-Order Commit Processor

Ali Hajiabadi
ali.hajiabadi@u.nus.edu
National University of Singapore
Singapore

Andreas Diavastos
andreas@ac.upc.edu
Universitat Politècnica de Catalunya
Spain

Trevor E. Carlson
tcarlson@comp.nus.edu.sg
National University of Singapore
Singapore

ABSTRACT

Modern superscalar processors execute instructions out-of-order, but commit them in program order to provide precise exception handling and safe instruction retirement. However, in-order instruction commit is highly conservative and holds on to critical resources far longer than necessary, severely limiting the reach of general-purpose processors, ultimately reducing performance. Solutions that allow for efficient, early reclamation of these critical resources could seize the opportunity to improve performance. One such solution is out-of-order commit, which has traditionally been challenging due to inefficient, complex hardware used to guarantee safe instruction retirement and provide precise exception handling.

In this work, we present Noreba, a processor for **N**on-speculative **O**ut-of-order **Re**tirement via **B**ranch Reconvergence **A**nalysis. In Noreba, we enable non-speculative out-of-order commit and resource reclamation in a light-weight manner, improving performance and efficiency. We accomplish this through a combination of (1) automatic compiler annotation of true branch dependencies, and (2) an efficient re-design of the reorder buffer from traditional processors. By exploiting compiler branch dependency information, this system achieves 95% of the performance of aggressive, speculative solutions, without any additional speculation, and while maintaining energy efficiency.

CCS CONCEPTS

• **Computer systems organization** → **Reduced instruction set computing**; • **Software and its engineering** → *Compilers.*

KEYWORDS

out-of-order commit, compilers, hardware-software co-design, processor design

ACM Reference Format:
Ali Hajiabadi, Andreas Diavastos, and Trevor E. Carlson. 2021. NOREBA: A Compiler-Informed Non-speculative Out-of-Order Commit Processor. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3445814.3446726

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446726

1 INTRODUCTION

Efficiency, in all forms, from energy-efficiency to area- and dollar-cost-efficiency, are extremely important in the post-Dennard Scaling, post-Moore's Law world that we live in today. But, the demand for efficient general-purpose processing, from datacenter servers to high-end mobile devices, continues unabated. While accelerators play a key role in addressing the efficiency (and performance) needs of specialized applications, the general-purpose processor continues to play a significant role in overall system deployment strategies. In fact, the traditional processor will be left with more difficult workloads, as the easy-to-process work is handed off to GPUs and other accelerators [4]. This means that more aggressive, as well as efficient strategies, are needed to advance general purpose core performance.

Core processor design has continued to push forward with more efficient, higher-performance processors. Typically, load instructions, specifically those that have been difficult to predict or compute, have been a focus of many attempts to improve performance. From traditional prefetching [13, 20, 30], execution-driven prefetching [18, 27], value prediction [23] and data recomputation [3], there have been many areas of active research that have been able to reduce the average memory access latency as seen by the processors themselves. In addition, there have also been techniques that have helped to initiate memory accesses earlier, or increase the amount of memory level parallelism (MLP) [12] that a processor can exploit. While much has been done to improve performance with high efficiency, it is the combination of delinquent loads and branches that depend on them that has held back the performance of the processor [31].

In order to improve overall performance, we need to keep reducing the average load latency seen, and enable making progress past the branches that depend on loads that are extremely difficult to predict. Loads, and the branches that depend on them, lock down the processor into a speculative state that solutions like value prediction cannot solve; without the data value from that delinquent load, the processor's speculative state will continue to remain large, limiting forward progress by the size of the processor's internal state. What is needed, instead, is the ability to continue to make non-speculative forward progress, even in the presence of load-dependent branches.

To address this issue, this work proposes the first energy-efficient implementation of non-speculative out-of-order commit processing, a compiler-assisted design that allows the processor to continue to make non-speculative forward progress, where none was possible before. By informing the hardware about which instructions are allowed to commit after a branch reconvergence point, the processor

can continue to make progress on independent work that is non-trivial to hoist above the original branch. Exposing this additional work allows the processor to issue additional loads, improving MLP, as well as commit instructions non-speculatively, freeing space in the processor to continue to make forward progress. By committing, and reclaiming resources of non-speculative, sure-to-be-committed instructions early and out-of-order, this technique aims to maximize the capabilities of the core, allocating resources to truly speculative instructions and state.

In this paper, we propose an energy efficient non-speculative out-of-order commit implementation that can reach 95% of the performance of a speculative, oracle (no commit mispredictions or roll-back penalties) out-of-order commit technique. This is on average a 1.22× improvement over an in-order commit baseline (up to a maximum of 2.17×), with just a 4% power overhead. To enable this:

- We implement a low-complexity compiler pass for branch dependency detection. We communicate this information to the hardware, to allow for the safe out-of-order commit of independent instructions.
- We propose a novel Selective ROB that implements out-of-order commit of instructions using low cost hardware. The Selective ROB allows for reordering to prioritize instructions that can commit early, without the additional hardware overheads typically seen in other solutions.
- We enable precise exceptions in the correct path of branch mispredictions with an efficient recovery mechanism that exposes out-of-order-committed instructions for OS handling of recovery or context switching.

In the rest of this paper, we first provide background on out-of-order commit in Section 2. In Section 3 we describe our compiler technique and in Section 4 we outline our architecture. In Section 5 we describe our experimental setup and in Section 6 we present our results. Finally, in Section 7 we outline state-of-the-art related works and conclude this work with Section 8.

2 BACKGROUND AND MOTIVATION

Bell and Lipasti [5] have investigated the necessary conditions that allow for a safe commit of an instruction, but none of the conditions imply that the instruction should be at the head of ROB. In other words, all instructions in ROB do not have to wait for the instruction at the head of ROB to resolve and commit which enables out-of-order commit of instructions. It is this insight which enables early out-of-order commit and reclamation. Briefly these conditions are:

- *Condition 1:* The instruction is completed;
- *Condition 2:* The instruction is involved in no memory traps. That is, we can't commit speculative loads and their dependent instructions;
- *Condition 3:* Register WAR hazards are resolved. We can't commit a write to a particular register unless all previous reads from that register are completed;
- *Condition 4:* There is no prior instruction that can raise an exception, like floating point instructions;
- *Condition 5:* All previous branches are successfully predicted and the instruction is in the correct execution path.

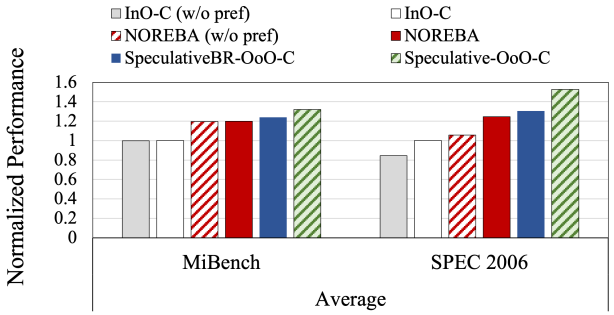

Figure 1: Performance improvement of different approaches over in-order commit for a Skylake-like processor baseline with prefetching.

If all these conditions are true for an instruction in the ROB, it can safely commit (*Non-Speculative OoO-commit*[1]). A more aggressive approach is to commit instructions even if one or more conditions are not preserved (*Speculative OoO-commit*). Processor designs using Speculative OoO-commit must be able to support checkpoint-and-rollback mechanisms in cases that the speculation fails. Implementing Speculative OoO-commit processing, however, can be expensive in matters of energy-efficiency; this is a significant concern for power-constrained processors. On the other hand, Non-Speculative OoO-commit designs do not need to support rolling back the architectural state of the system. Hence, they can offer a better level of energy efficiency. However, current non-speculative designs are unable to realize the full potential of OoO-commit to achieve high-levels of performance.

Figure 1 shows the performance of various OoO-commit solutions normalized to a traditional in-order commit design in a Skylake-like processor[2]. Speculative OoO-commit is presented as an oracle upper-bound, therefore it doesn't include any penalties for misspeculation. *SpeculativeBR OoO-commit* preserves all OoO-commit conditions except for the branch condition (*Condition 5*). This solution commits instructions even if previous branches have not yet been resolved. Its performance shows a large potential for improvement when relaxing the branch condition as it achieves on average 86% of the oracle Speculative OoO-commit.

The key insight of this paper is that it is possible to relax the branch condition (*Condition 5*) and commit non-speculative instructions early and out-of-order in an energy-efficient way. To achieve this, we adopt a hardware-software co-design solution that uses the compiler to expose instruction dependencies on earlier branches (not necessarily the most recent branch instructions) to the processor that is equipped with new energy-efficient structures that use this information to enable non-speculative out-of-order commit. The compiler has all the information on possible execution paths and can make early decisions about control and data dependencies

[1]Since the publication of Bell et al.'s work [5], recent research [16, 24] has shown that Conditions 1 and 3 can be relaxed in a non-speculative way in specific cases. For example, instructions can be committed even if the results have not returned if we can guarantee that execution will continue (Conditions 2, 4 and 5 hold). In this work, we define Non-Speculative OoO-commit as the ability to recover resources non-speculatively, and therefore we do not follow these original conditions strictly (In this work, Conditions 1 and 3 do not need to hold if all other conditions are met).
[2]We use C/C++ subset of SPEC CPU2006 benchmark suite

in a Non-Speculative way that relieves the processor from expensive checkpoint-and-rollback mechanisms. Our hardware design employs small, simple structures in the implementation, that improve the power efficiency over prior OoO-commit solutions, such as a collapsing ROB [5, 11].

Our proposed solution relaxes the branch condition without introducing additional speculation by using compiler information and unlocks the performance gains of SpeculativeBR OoO-commit, while also maintaining the power efficiency of in-order commit designs by avoiding complex hardware structures like collapsing commit buffers.

3 BRANCH DEPENDENT CODE DETECTION PASS

We use the compiler to perform control-flow and data-flow analysis and mark true dependent instructions for each branch. Marking true branch dependencies and exposing this information to the hardware allows the instructions to commit and release their resources as early as their true dependencies are resolved. In other words, instructions not marked as dependent on a branch are candidates for OoO-commit, if that branch is still unresolved and blocking the head of ROB. Note, that the instructions might still be dependent on an earlier branch. Effectively, we assign a single branch, either the most recent, or an older branch, as the dependent branch for each instruction. The NOREBA compiler pass, branch dependent code detection, consists of four steps as shown in Figure 2, with an if-then-else structure represented by basic blocks:

Ⓐ Detecting the branch reconvergence point: The reconvergence point of a branch is the earliest instruction in the subsequent instructions of the program that we expect the control flow will eventually reconverge to, regardless of the branch outcome. We find the reconvergence point by determining the immediate post-dominator of the branch in the control-flow graph [7, 29]. In Figure 2, label `L2` is the reconvergence point of the branch.

Ⓑ Detecting control dependent instructions: We perform reachability analysis to detect branch control dependent instructions. All basic blocks that can be executed between the branch and its reconvergence point are control dependent on the branch. Since it is guaranteed that the program control will reach the reconvergence point after executing the branch, we can easily traverse all basic blocks between the branch and the reconvergence point by following the successors of the branch until we reach the reconvergence point. Note, that if an instruction is control dependent on multiple branches (like nested loops), we consider the instruction to be dependent on the inner most branch. In Figure 2, all instructions in BB2 and BB3 (the red area) are control dependent on the branch, since their execution depends on the outcome of the branch.

Ⓒ Detecting data dependent instructions: Since we would like to commit independent instructions out-of-order, we need to make sure that these instructions have no data dependencies with the branch and the control dependent instructions. We consider a set of instructions to be data dependent if their values might be different based on the path executed after the branch. We detect data dependent instructions by analyzing the def-use chains and memory aliasing of variables. Instructions using the values from

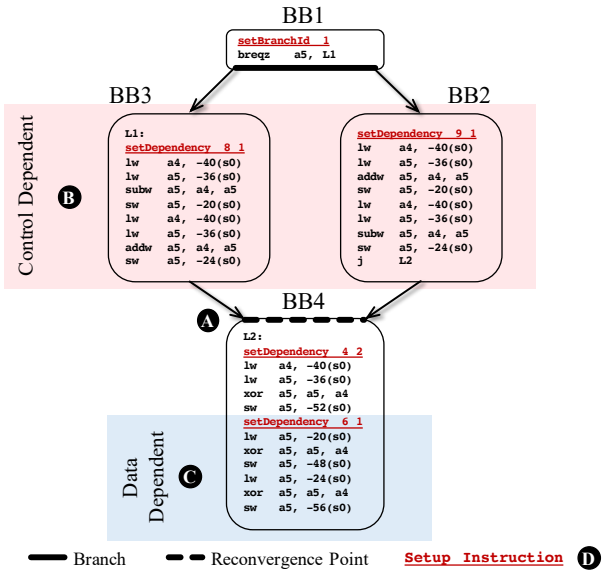

Figure 2: A simple if-then-else structure represented with basic blocks marked by the branch dependent code detection pass. Branch ID 1 is assigned to the branch in BB1 with the `setBranchId` instruction, and all instructions in the red and the blue areas are control- and data-dependent respectively on this branch. We mark these regions with the `setDependency` instruction. In BB4, the first 4 instructions are independent from this branch, but are marked as dependent on an earlier branch, with branch ID 2.

control dependent instructions or must/may alias with control dependent instructions are considered data dependent. In the example of Figure 2, both BB2 and BB3 are updating `-20(s0)` and `-24(s0)` locations. As you can see, the first four instructions in BB4 are independent from the updates in BB2 and BB3, but the data of the next 6 instructions (the blue area) are dependent on the execution of BB2 or BB3.

Ⓓ Marking branch dependent regions: At this point, we have a set of control- and data-dependent instructions for each branch. We mark branches and their dependent regions using two new setup instructions: (1) `setBranchId` which assigns an ID to the branch, and (2) `setDependency` which marks the consecutive instructions dependent on the same branch (refer to Table 1 to see the format of setup instructions). In Figure 2, we mark the branch with `setBranchId` and assign 1 as the ID to this branch. Then we mark all the regions detected as dependent in previous steps with `setDependency` instructions. Note, that the first instructions in BB4 are independent from the branch in BB1, but they might be dependent on an earlier branch, for example a branch with ID equal to 2.

Limitations of compiler-only solutions. Step Ⓓ in the branch dependent code detection pass can be replaced by a compiler hoisting pass that reorders instructions based on the analysis of steps Ⓐ, Ⓑ, and Ⓒ, in a way that only truly dependent instructions exist after the branch and independent ones are hoisted before the

```
for (i=0; i<rarp.elemqu; i++){
    rarp[i]->centerp.x=0;
    rarp[i]->centerp.y=0;
}

for (y=0; y<=mapmaxy; y++){
    for (x=0; x<=mapmaxx; x++){
        regionp=regmapp(x,y);
        if (regionp){
            regionp->centerp.x+=x;
            regionp->centerp.y+=y;
        }
    }
}
```

Listing 1: Two critical independent `for` loops in `astar` that a compiler analyzer cannot statically decide the best ordering.

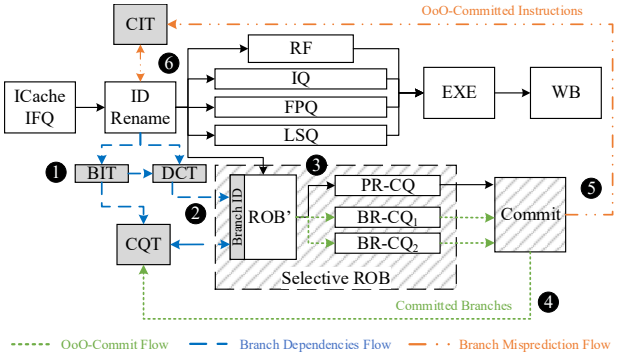

Figure 3: The Noreba core microarchitecture.

branch. However, reordering independent regions of code is not a trivial task for a compiler analyzer with no dynamic information. Take for example the `astar` application from SPEC CPU2006 and the two consecutive `for` loops in Listing 1. The two loops are independent of each other and can be reordered without violating the correctness of the code. Our analysis shows that the two branches associated with the two `for` loops have the same criticality. However, a compiler analyzer cannot statically determine which order of the loops will produce the highest performance, thus will opt to not reorder them. In Noreba, independent instructions that are executed will be committed, regardless of the order of the loops, and critical processor resources will be released allowing for more work to be fetched in the processor and performance to be improved (see Section 6). The Selective ROB design we describe in Section 4 allows for instructions dependent on different branches (like the two loops in this example) to be separated and committed independently.

The highest performing ordering of instructions is affected by the criticality of loads and branches and in turn their criticality is affected by the program inputs and the processor microarchitecture. Therefore, a compiler analyzer requires runtime information to successfully implement a high performance instruction ordering. One such solution is Profile-Guided Optimization (PGO) where the underlying hardware provides the compiler with execution-based feedback. However, PGO is not always practical because the optimizations provided are highly dependent on the input of the application and the load or utilization of the machine during the profiling. PGO also has some inherent limitations: (1) the complexity of the optimization process increases exponentially with the number of independent code blocks in the source code, and (2) hoisting instructions across basic blocks can have user visible side-effects and might violate the semantics of the source program [26]. In a hardware implementation, like Noreba, such problems don't exist since the semantics of the code can be honored, while reordering the instructions to improve performance.

4 NOREBA MICROARCHITECTURE

By collecting branch dependency information during the branch dependent code detection pass, the Noreba core can relax the Branch OoO-commit condition (Condition 5). It allows instructions beyond the reconvergence point of a branch to safely commit if they do not depend on any previous speculative (unresolved) branch. Branch dependency information collected at the software level is propagated to the core by extending the Instruction Set Architecture (ISA) with two new instructions. These instructions are injected in the code by the compiler to uniquely mark branches and identify dependent instructions on those branches. The `setBranchId` instruction is inserted before every marked branch in the code and assigns a unique identifier (ID) to it. Instruction dependencies on marked branches are set with the `setDependency` instruction that is inserted at the beginning of a block of dependent instructions and defines a number of consecutive instructions that follow and the ID of the branch they depend on. While these new instructions occupy fetch slots in the Instruction Fetch Queue (IFQ), they will be dropped at the decode stage and will not be sent to the execution units (in a similar way to how move instructions are handled in Intel processors [14]).

We identify three main process flows for the proposed Noreba core microarchitecture described in Figure 3: (1) the Branch Dependencies Flow, marked in blue dashed lines, that propagates information from the compiler to the hardware (Section 4.1), (2) the OoO-Commit Flow, marked in green dotted lines, that commits instructions out-of-order where possible (Section 4.2) and (3) the Branch Misprediction Flow, marked in orange dashed lines, that allows the processor to correctly recover from branch mispredictions event (Section 4.3). In Section 4.4 we describe in detail how exceptions are handled in the proposed implementation.

Figure 4 shows example data in the new structures of the proposed architecture and Table 1 shows a detailed Event-to-Action analysis of the Branch Dependencies and OoO-Commit Flows.

4.1 Branch Dependencies Flow

With each marked branch, the `setBranchId` instruction stores the compiler-defined ID of the branch with its unique sequence number in the Branch ID Table (BIT) (step ❶ in Figure 3). When a `setDependency` instruction is decoded, it stores in the Dependents Counter Table (DCT) the number of consecutive instructions that follow and depend on a specific branch. At any given moment, the DCT only holds one counter for one branch with its dynamic BranchID that is calculated as a tuple of the compiler-defined ID and the sequence number of the branch (queried from the BIT). In step ❷, any instruction entering the ROB checks the DCT counter

Branch ID Table (BIT)

ID	Sequence Number
01	1015456
10	1023358
11	1028742
…	

Commit Queue Table (CQT)

BranchID [ID, Seq. Num]	BR-CQ
[01, 1015456]	0
[10, 1023358]	1
[11, 1028742]	0
…	

Dependents Counter Table (DCT)

BranchID [ID, Seq. Num]	Counter
[11, 102842]	3

Committed Instructions Table (CIT)

PC	Most Recent Branch (ID)	Register Mapping
462868	10	$r27
462884	10	$r13
46288a	11	$r2
…		

Figure 4: The Branch ID Table (BIT) identifies compiler-marked branches with their dynamic sequence number, the Commit Queue Table (CQT) stores the Commit Queue a branch was steered to in the Selective ROB, the Dependents Counter Table (DCT) stores the number of subsequent instructions that depend on a specific branch and the Committed Instructions Table (CIT) is used to avoid re-execution of out-of-order committed instructions in an exception event.

and if there is a non-zero value, a dependency is marked by assigning the DCT.BranchID to the instruction's BranchID position in the ROB as a 3-bit entry. At the same time, the DCT Counter is decremented by 1. For instructions that are not marked as dependent (either they are completely independent of all previous branches, or instructions from a program that was not compiled with our compiler infrastructure) we reserve BranchID 0 in the corresponding ROB field.

4.2 OoO-Commit Flow

State-of-the-art OoO-commit solutions use either an associative ROB or a collapsing ROB implementation to support OoO-commit of instructions [5, 11]. However, such implementations tend to require complex and power-consuming hardware that could limit the efficiency gains of the processor. Therefore, we propose a multi-queue ROB design, called the Selective ROB (or ROB') that allows for the reordering of instructions to prioritize non-speculative (their dependent branch has resolved) instructions. Decoded instructions are inserted at the tail of ROB' in program order, while instructions at the head of ROB' are steered to specialized FIFO Commit Queues (CQ) according to their branch-dependency state:

(1) *Branch-independent instructions:* If an instruction is independent of any live branches in the Commit Queue Table (CQT), it will be steered to the Primary Commit Queue (PR-CQ) to be committed as soon as it arrives at the head of the queue and satisfies all other commit conditions. Note that, instructions of a program that isn't compiled with our compiler infrastructure fall into this category. Such instructions will be send to PR-CQ in program order and will commit in program order;
(2) *Branch-dependent instructions:* If an instruction is dependent on a live branch from the CQT, it will be steered to the appropriate Branch Commit Queue (BR-CQ) to wait for its branch to resolve before becoming eligible for commit.

Table 1: Detailed OoO-commit processor events and actions.

Event	Action
❶ `setBranchId ID` decoded and branch entering ROB'	BIT[ID] = Branch Sequence Number
❶ `setDependency NUM ID` decoded	DCT.BranchID = (ID, BIT[ID]) DCT.Counter = NUM

❷ Any instruction entering ROB'	if DCT.Counter > 0:
	  Inst.BranchID = DCT.BranchID
	  DCT.Counter = DCT.Counter - 1
	else:
	  Inst.BranchID = INVALID

❸ The branch exiting ROB'	CQT[BranchID] = CQ Steer branch to CQ

❸ Any instruction exiting ROB'	if CQT[Inst.BranchID] exists:
	  Steer instruction to CQT[Inst.BranchID]
	else:
	  Steer instruction to PR-CQ

To reduce commit stalling due to unresolved branches that are independent from one another, we adopt a multi-queue implementation for BR-CQs (two BR-CQs in this implementation). Instructions are steered to a BR-CQ based on their dependency to an unresolved branch (step ❸). An unresolved branch instruction at the head of the ROB' is steered to one of the BR-CQs and updates the corresponding Commit Queue (CQ) entry in the CQT. Subsequent instructions match their branch dependencies with the unresolved branches in CQT and are steered to the same queue. To ensure that load and store instructions are committed in program-order, they are steered to the CQs once they are resolved in the Load/Store unit and their page-table access succeeds.

Due to the low power consumption of a queue-based ROB (see Figure 16), we choose the size of the ROB' to be equal to the size of the ROB of the baseline core. Even-though instructions can commit out-of-order, the total commit capacity (width) of the processor remains the same as the baseline processor. More details on the system configuration can be found in Table 2, with a detailed performance characterization on queue sizes in Section 6.1.1. Finally, committed branch instructions remove their entry from the CQT as they are no longer required (step ❹).

4.3 Branch Misprediction Flow

To recover from a branch misprediction event, the processor flushes the pipeline and fetches instructions from the correct execution path, including instructions beyond the reconvergence point of the branch. However, instructions beyond the reconvergence point that committed out-of-order should not re-execute. To identify these instructions and discard them in the front-end, we use a Committed Instructions Table (CIT) that stores instructions (with their register mapping information) that were recently committed out-of-order. In step ❺, an entry in the CIT is allocated for every instruction that is committed out-of-order. The updated Commit unit of the processor recognizes instructions that committed out-of-order using the instruction PC. After a branch misprediction, a CIT hit when re-fetching instructions (step ❻) identifies instructions that have already committed, thus the re-fetched instruction is

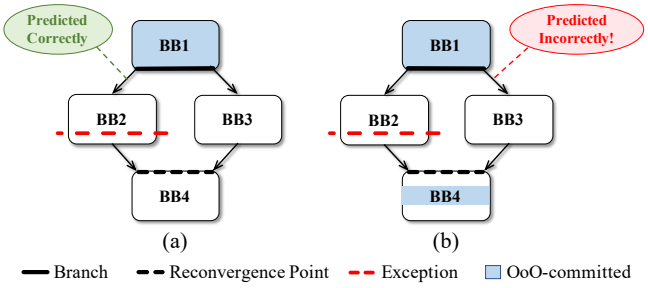

Figure 5: Two general cases of exceptions in Noreba.

dropped at the Decode stage. At the same time the output register is marked for consuming instructions to know where the value will be found and the CIT entry is rerouted through the OoO-commit processor and reinserted in the CIT in case a new misprediction happens again. Note, instructions are identified based on their PC.

Finally, CIT entries are freed by the Commit unit according to the commit sequence of instructions. With every OoO-committed instruction in the CIT we also store the ID of the most recent unresolved branch. The commit of the most recent unresolved branch at the time an instruction commits out-of-order, guarantees that all branches that the specific instruction beyond their reconvergence point have been committed and it is safe to clear its entry from the CIT.

4.4 Precise Exception Handling

We build the Noreba processor on top of the RISC-V ISA [32], which limits exceptions to floating-point and memory-related events. In RISC-V based systems, we can use the floating-point control and status register, `fcsr`, to accrue the exceptions that occur during floating-point operations. In the normal execution of a RISC-V processor, floating-point exceptions do not cause traps. However, it is possible to add software checks in the code to examine the `fcsr` register and call an exception handler based on its value. We do not include software checks in the code for floating-point exceptions but it is common to report the status at the end of execution, and software developers can check when desired [1].

Regarding memory-related exceptions, the exception can arise in both the correct path of a correct branch prediction and also the correct path of a branch misprediction (see Figure 5). Our Noreba implementation waits for the success of the page-table access through the TLBs before proceeding to out-of-order commit instructions beyond the memory operation. However, memory exceptions arising in the correct path of a branch misprediction can be more complex for precise exception handling since we might have already committed some instructions out-of-order beyond the reconvergence point (Figure 5b). In this case, we have to make sure to inform the OS of the changes the OoO-committed instructions have made in the processor's state and also, successfully resume OoO-commit after coming back from the OS. When we switch to the OS at the point of an exception, we expose the CIT information to the OS, since CIT contains all the recent instructions committed out-of-order beyond the branch reconvergence point and what changes they have made (register mappings). We introduce two new instructions to communicate between CIT and the OS. The OS can use `getCITEntry` instruction to iterate over all of the entries in the CIT and store them, and use `setCITEntry` to load the stored entries in the CIT. The OS can use these instructions as it is required for exception handling, context switching, etc. in order to make sure we recover from the exception and successfully resume execution of the application running in OoO-commit.

Note, that all types of exceptions in a RISC-V system (like page faults caused by illegal address accesses, `mprotect`-style accesses, illegal instructions, etc.) can also be handled in the same way as described above.

4.5 Noreba Multi-Core

To deploy the Noreba architecture in a multi-core design, we need to take into account how OoO-commit might affect data and resources sharing. Here, we discuss the three main aspects that we need to take into consideration.

Race Conditions. To guarantee well-synchronized and Data Race Free (DRF) programs we employ the compiler to detect the synchronization barriers in the application and perform the Noreba compiler pass only between those barriers. Instructions at these boundaries are marked as dependent, thus forcing them to commit in-order.

Memory Consistency. Because the Noreba architecture operates in a relaxed memory model, only the memory barriers need to guarantee consistency. This information is statically available, thus the compiler can guarantee that the processor will avoid OoO-commit at the memory barrier boundaries.

Cache Coherence. TLB accesses are checked in program order before steering instructions to the commit queues and the data is guaranteed to return (refer to Section 4.2, and [16]). Thus, cache coherence is not affected by our design.

5 EXPERIMENTAL SETUP

Simulation Environment. To evaluate the performance of this work, we use the gem5 simulator [6] in syscall emulation mode. We implemented the OoO-commit microarchitecture on top of the O3 core. For power analysis we modified McPAT [22] version 1.3, to support our microarchitecture. As a baseline core, we use a Skylake-like processor. Also, we use Delta-Correlating Prediction Tables (DCPT) as a prefetcher in our baseline simulations [13]. Detailed processor parameters are shown in Table 2 and Table 3.

Compiler Implementation. We use LLVM-10.0 [21] for our compiler analysis. We implement Branch Reconvergence Detection in machine-level for RISC-V architecture. However, our pass is not architecture-dependent and can be registered at the LLVM back-end for any other architecture.

Benchmarks. We target the C/C++ application subset[3] of the SPEC CPU2006 [19] and MiBench [15] benchmark suites in our evaluations. For SPEC CPU2006 suite, we use gem5 to collect Basic Block Vectors (BBVs) and use SimPoint [17] to find the representative 1 B instruction region. Also, we use reference inputs for checkpointing

[3]The RISC-V backend of the LLVM 10.0 compiler does not currently support Fortran applications, but we expect to see even greater benefits for Fortran applications as seen in prior work [24].

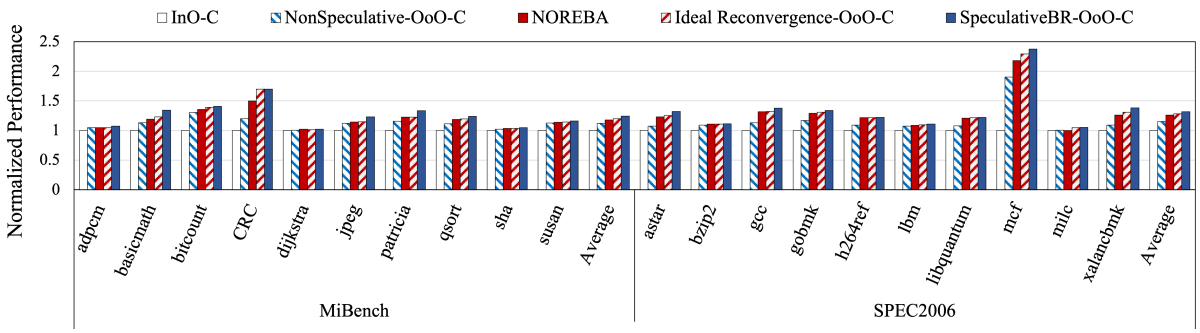

Figure 6: Performance of various OoO-commit modes normalized to in-order commit (InO-C) on a Skylake-like processor.

Table 2: System configuration.

L1d Size	32KB, 4clk
L1i Size	32KB, 4clk
L2 Size	256KB, 12clk
L3 Size	1MB, 36clk
Dispatch/Issue/Commit Width	4/4/4
Branch Predictor	TAGE-SC-L-8KB
Prefetcher	DCPT
Selective ROB	
ROB' entries	baseline core ROB
BR-CQs entries	2 × 8-entries
PR-CQ entries	8-entries
BIT/CQT entries	8
CIT entries	128

Table 3: Baseline microarchitecture configurations.

Microarchitecture	ROB	IQ	LQ/SQ	RF
Nehalem-like (NHM)	128	56	48/36	64
Haswell-like (HSW)	192	60	72/42	128
Skylake-like (SKL)	224	68	72/56	168

and building representatives. From the MiBench suite, we simulate the entire applications.

6 EXPERIMENTAL RESULTS

To better understand the performance and energy efficiency of Noreba, we perform an extensive evaluation of the proposed processor and present our findings in this section. We compare the results of Noreba with the performance of various other out-of-order commit implementations. All reported average numbers are calculated based on the Geo-mean of all applications' execution runtime.

6.1 Performance Evaluation

Figure 6 shows the performance improvement of various OoO-commit approaches over a baseline in-order commit processor. We evaluate four different approaches:

- **NonSpeculative-OoO-C**. Instructions are committed out-of-order once they satisfy all commit conditions; *i.e.* all previous branches and load instructions are resolved [5].
- **Noreba**: We use the compiler to mark branch-dependent regions of instructions, and implement the Selective ROB structure in the processor to commit Non-Speculative branch-independent instructions out-of-order.
- **Ideal Reconvergence-OoO-C**: Instructions are committed based on the same compiler branch dependency information as Noreba, but without hardware restrictions, using an ideal ROB implementation that allows for arbitrary reordering of instructions based on the optimal commit sequence.
- **SpeculativeBR-OoO-C**. Instructions are committed speculatively with respect to unresolved branches and an ideal ROB implementation with arbitrary instruction reordering. This represents the upper bound of any solution that relaxes the branch condition (Speculative and Non-Speculative). However, a SpeculativeBR OoO-commit implementation will need to roll back to a valid state if speculation fails and this cost is not accounted for in the results presented in Figure 6.

Results in Figure 6 show that across all applications, Noreba achieves on average 95% of the performance of the upper-bound SpeculativeBR OoO-commit and 1.22× performance improvement over the conservative in-order commit implementation.

To understand where the performance improvement is coming from, we study the behavior of critical branches (frequently cause the ROB to stall before they resolve) of `bzip2` and `mcf` that achieve the lowest and highest improvement respectively. In Figure 7, the x-axis plots the number of instructions that are dependent on a branch, while y-axis plots the number of cycles that the same branch caused the ROB to stall. Branches in `mcf` take longer to resolve but have fewer dependent instructions present in the ROB and yet independent instructions are delayed for longer periods of time. In other words, there are many instructions in the ROB that are independent from the branch that stalls the ROB and they can be committed without waiting for the branch to resolve. `bzip2` on the other hand, has a larger number of instructions the depend on the stalling branch and cannot be committed until the branch is resolved.

Figure 8 shows the percentage of dynamic instructions committed out-of-order using Noreba. Applications with little to no

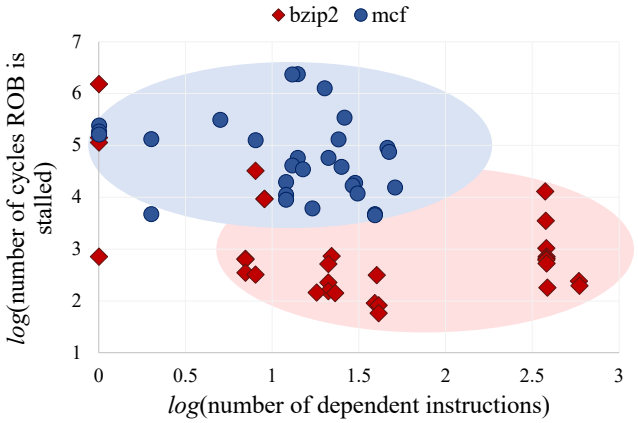

Figure 7: Distribution of critical branches for `mcf` and `bzip2`. The majority of branches in `mcf` (blue area) are stalling the ROB for more cycles than branches in `bzip2` (red area) do. There are also fewer dependent instructions waiting for the branch to resolve. This means that in `mcf` there are more independent instructions ready to commit while the branch is stalling the ROB, hence the large performance improvement in `mcf`. The baseline core is a Skylake-like processor.

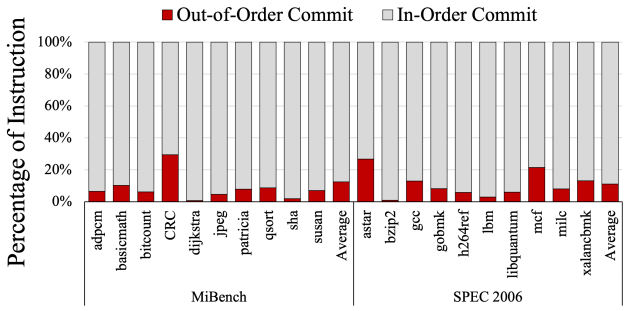

Figure 8: Dynamic instructions committed out-of-order in Noreba. The baseline core is a Skylake-like processor.

performance improvement, over an in-order commit design, like `bzip2` and `dijkstra`, commit very few instructions out-of-order. Applications with high number of independent instructions beyond the reconvergence point of a branch (*i.e.* `CRC` and `mfc`) commit more than 20% of their instructions out-of-order, hence the performance improvement we observe in Figure 6.

6.1.1 Evaluating the Size of Selective ROB. The number and size of the Branch Commit Queues (BR-CQs) we choose were selected based on the offered performance and the power consumed in order to maintain an energy-efficient package. Figure 9 shows that for a traditional 224-entry ROB, the performance of Noreba saturates at 2 BR-CQs with 8 entries each. These options allow Noreba to achieve 99% of the performance of an Ideal Reconvergence OoO-commit processor with the same number of entries (Skylake-like processor). Figure 10 presents the power consumption of the same options and the results show that the simplicity of the Selective

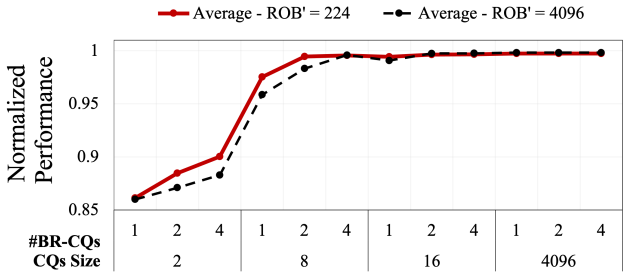

Figure 9: Normalized performance of different selective ROB configurations for a Skylake-like processor with two different ROB' sizes. Performance is normalized to the ideal Reconvergence-OoO-C with the same ROB size.

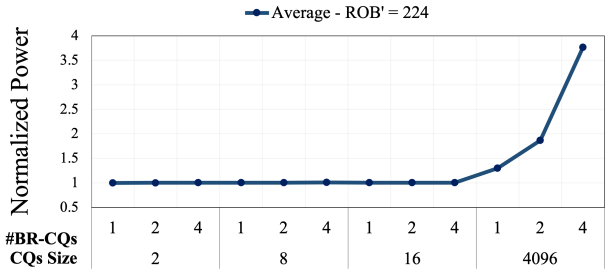

Figure 10: Normalized power of different selective ROB configurations for a Skylake-like processor. Power is normalized to the minimum configuration.

ROB design allow it to be scalable to larger sizes (if needed) with very small effect on power consumption. The power consumption only grows to prohibitive values after a significantly large number of BR-CQs entries that goes well beyond what is necessary to enable improved performance. For this work, we choose 2 BR-CQs with 8 entries each.

6.1.2 Impact of Setup Instructions on Performance. To inform the hardware of the compiler findings on instruction dependencies, Noreba employs a set of setup instructions. Because our compiler pass inserts a single setup instruction per region to identify dependent instructions, applications with many dependent instructions that are fragmented and reside in multiple consecutive regions are prone to higher performance overheads. However, Figure 11 shows that the overhead on performance by the added instructions is on average only 3% compared to a perfect design that does not require the use of setup instructions.

6.1.3 Sensitivity to the Size of Resources. Applications in the SPEC CPU2006 suite scale their performance with larger core resources provided by more aggressive microarchitectures, both for in-order and out-of-order commit techniques. And as Figure 12 shows, the same applies for Noreba. For this study we consider three different core sizes: *Nehalem-like* processor (NHM), *Haswell-like* processor

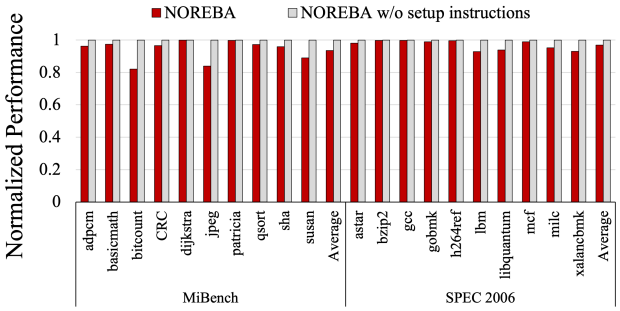

Figure 11: Impact of inserting setup instructions in order to specify branch dependent regions in the code.

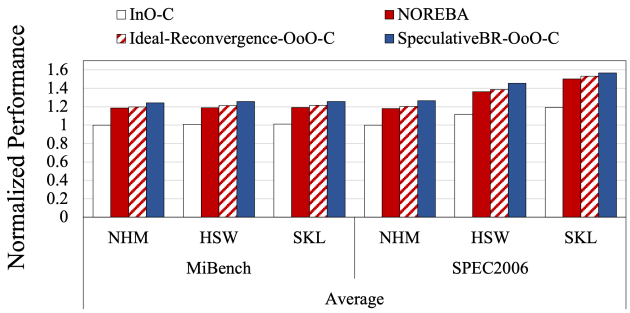

Figure 12: Performance for different core designs.

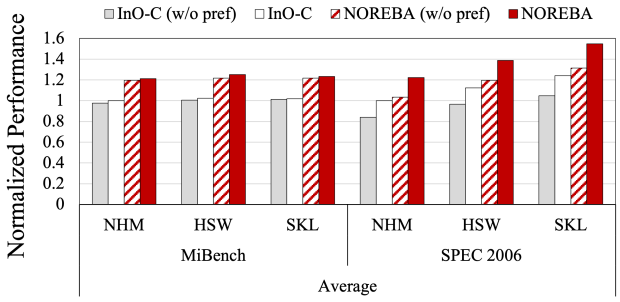

Figure 13: Effectiveness of prefetching on the Noreba core. Performance is normalized to the NHM in-order commit with prefetching.

(HSW), and *Skylake-like* processor (SKL). The architectural details of each core are described in Table 3.

6.1.4 Evaluating the Impact of Prefetching. Prefetching helps the processor to reduce the penalty of long latency loads and consequently reduce the stall time of load-dependent branches at the head of the ROB. Thus, successful prefetching allows instructions to become commitable much earlier and in combination with an out-of-order approach the processor resources are released earlier to accommodate for more future instructions. Figure 13 shows how Noreba is even more effective when combining prefetching with OoO-commit.

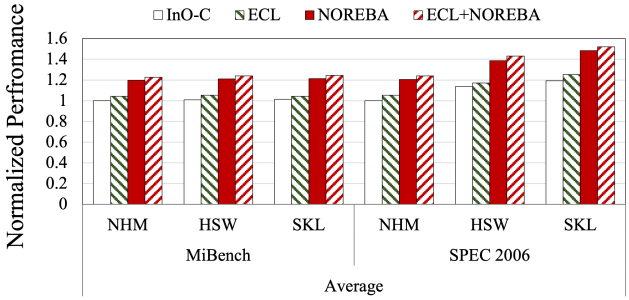

Figure 14: Early Commit of Loads (ECL) on in-order commit processor and Noreba. The baseline core is a Skylake-like processor.

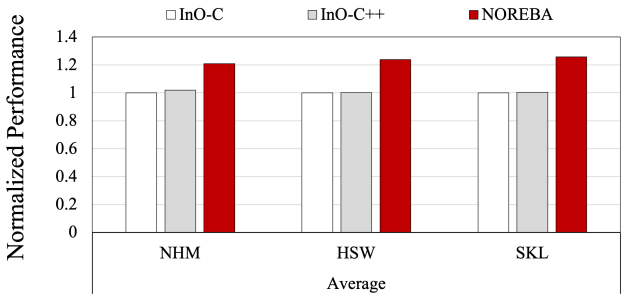

Figure 15: Performance with increased commit bandwidth (InO-C++: In-order commit with 8 instructions commit width).

6.1.5 Evaluating the Impact of Early Commit of Loads. Early Commit of Loads (ECL) allows for load instructions to commit as soon as they are guaranteed not to cause an exception [16]. Such a technique alone offers modest performance gains but the same performance benefit we get on an in-order commit processor is achieved on Noreba. Figure 14 shows the extra performance benefit offered by ECL on Noreba.

6.1.6 Evaluating the Impact of Commit Bandwidth. With the results presented in Figure 15 we want to highlight that increasing the number of instructions that can be committed per cycle is not enough to improve application performance on a traditional in-order-commit processor. Out-of-Order commit solutions take advantage of available core resources (emulating a larger ROB, reclaiming resources early through early and out-of-order commit, etc.) to allow not only instructions to commit earlier but more instructions to enter the processor and execute earlier.

6.2 Power and Area Overheads

Figure 16 shows the power and area overheads of the proposed implementation normalized to the baseline in-order commit processor. Overall, we see an average power overhead of 4% that comes from the implementation of the Selective ROB and the added CIT, CQT, and BIT structures that implement the Noreba design. The queues

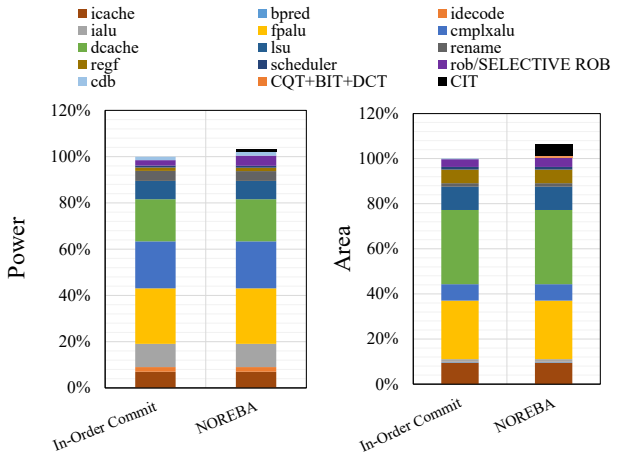

Figure 16: Total power and area consumption normalized to the In-order commit baseline processor. The new structures are in capital letters in the legend.

in the Selective ROB only marginally increase power as they are implemented as simple First In First Out (FIFO) queues. The CIT, CQT, and BIT tables are direct-mapped memory structures that also require little power. The average area overhead produced by the new structures is approximately 8% over the baseline in-order commit core.

7 RELATED WORK

Previous OoO-commit processors can be categorized into (1) Speculative, and (2) Non-speculative solutions. Speculative solutions allow higher flexibility in committing instructions to produce high levels of performance, but require checkpoint-and-rollback mechanisms to revert the architectural state to the original state in the case of misspeculation. Such mechanisms have been traditionally implemented using inefficient hardware structures that limit the processor's energy efficiency. Non-speculative solutions implement a Safe OoO-commit design that doesn't allow committing speculative instructions. Therefore, expensive check-pointing or rollback mechanisms that correct the architectural state of the processor are not required. However, as we have shown, prior non-speculative designs do not take advantage of the full potential of an OoO-commit processor.

Compiler support has also been used in OoO-commit solutions, where the program is statically modified to increase OoO-commit eligible instructions. However, previous solutions require complex and power consuming core structures to implement the out-of-order commit of instructions. To the best of our knowledge, NOREBA is the first to use the compiler to identify independent instructions beyond the reconvergence point of a branch. Unlike other OoO-commit processors, we use energy-efficient structures to implement a non-speculative out-of-order commit design that uses compiler information to make progress based on true branch dependencies.

Table 4 summarizes previous state-of-the-art solutions for OoO-commit designs and outlines their basic characteristics and their proposals on the hardware implementation.

Speculative OoO-commit solutions. A processor that assumes it is following the correct path in every branch instruction, without waiting for it to resolve, can commit many instructions and release its resources earlier, offering high potential for performance improvement. However, speculatively committing instructions doesn't necessarily make forward progress, as the processor would have to return to the correct path in case of a misspeculation. To do so, expensive roll back mechanisms are required.

In [25], instructions are committed in program order, but release critical resources, like RF and LQ/SQ, early to allow for more instructions to be processed in parallel. However, not committing instructions reserves ROB slots that limits the amount of work the processor can do. In [2], the authors increase the commit granularity of the ROB from a single instruction to a group of instructions that write to the same register. A group's end is marked by the last instruction that writes to a shared register. Special instructions are used and additional entries are reserved to mark the start and end of each group in the ROB. In addition, a checkpoint is marked at the beginning of each group to revert changes made in case of a misspeculation event. The authors of [9] remove the conventional ROB structure and replace it with checkpoint-oriented mechanism that allows committing instructions in groups by adding checkpoints in specific points of the program. Assuming all instructions are executed and no misspeculation arises, resources are released. In case of a misspeculation, the processor rolls back to the previous checkpoint and restarts.

In [10], the authors propose a speculative OoO-commit implementation for in-order processors. They use the compiler to divide the program into idempotent regions and this enables the processor to recover from misspeculations just by re-executing the idempotent regions, hence there is no need to store hardware checkpoints. Their proposal targets in-order processors and the benefits of their OoO-commit implementation is limited to the size of idempotent regions that the compiler identifies. Large idempotent regions can lead to the re-execution of a large number of instructions. NOREBA avoids any speculation and there is no need for any hardware checkpoints or re-execution of instructions.

Non-speculative OoO-commit solutions. Committing instructions in a safe, non-speculative way allows for efficient allocation of processor resources that provides for correct execution without requiring expensive check-pointing and rollback mechanisms. However, expensive mechanisms are still required to find safe-to-commit instructions in the ROB.

Bell and Lipasti in [5] defined the necessary conditions that allow an instruction to safely commit, even if it's not at the head of the ROB. To implement this, they propose a collapsing ROB, that arbitrarily removes entries from the ROB and either collapses it to fill the void created or manage the gaps on a free list to determine where new entries can be added. In [24], the authors replace ROB with a Validation Buffer that uses speculative instructions (*i.e.* branches) to create groups of instructions called epochs. A Validation Buffer waits only for epoch initiators at it's head to resolve and releases all instructions in the preceding epoch.

Table 4: State-of-the-art solutions and their hardware implementations for out-of-order/early commit/resource reclamation. *Our main results do not include Early Commit of Loads (ECL) but NOREBA can support this feature as shown in Section 6.1.5.

Paper	Non-Speculative	Compiler-Assisted	Early commit	OoO-Commit	Early reclamation	Hardware Implementation
Deconstructing Commit [5]	✓					Collapsing ROB
Validation Buffer [24]	✓		✓			Validation Buffer
Cherry [25]					✓	Normal ROB/ Early release of RF and LSQ
Compiler Assisted OoO-commit [11]	✓	✓		✓		Collapsing ROB
DeSC [16]	✓	✓	✓			Normal ROB/ Decoupled design/ Early commit of terminal loads
OoO-commit Processors [9]				✓		Check-pointing ROB
Group Commit [2]				✓		Augmented ROB
Idempotent Processor [10]		✓		✓	✓	In-order processor with Slice Data Buffer (SDB) and non-blocking cache
NOREBA	✓	✓	✓*	✓	✓	Multi-queue selective ROB

Some non-speculative designs use the compiler to assist the hardware to decide which instructions can commit early. DeSC [16] uses the compiler to decouple the program into memory access and value computation parts and run the program in a decoupled hardware design. They detect and commit early loads that are only for computational purposes (terminal loads). In [11], the authors use the compiler to choose a commit mode for instructions. The compiler partitions the code into blocks that can either commit in-order of out-of-order.

Branch reconvergence detection techniques. Finding the reconvergence point of a branch allows us to specify control- and data-independent instructions beyond that point. Previous work in this direction has used hardware-only solutions to predict reconvergence points [7, 8, 28, 29]. However, these hardware-based solutions work in a speculative way that doesn't allow for a Safe OoO-commit design. In our work, similarly to [28], we use the compiler to specify independent instructions and enable out-of-order-commit of instructions in a safe and efficient way.

Most of the solutions presented in this section require an associative ROB structure to search for instructions that satisfy commit conditions. However, such a structure consumes large amounts of power, making it unsuitable designs for energy-efficient processors.

8 CONCLUSION

General-purpose processors continue to be the work-horse for many workloads, from mobile and embedded devices, to server-class systems. But, recent performance improvements of processor cores have been modest, at best, as processor technology has failed to continue to provide the continued cost reductions and transistor power budget of years past. To overcome these challenges, architects are looking for promising, efficient solutions that can continue to provide performance for general-purpose applications, in an energy efficient way.

In this work, we propose the first non-speculative out-of-order commit and reclamation implementation, that combines compiler-directed annotation of independent instructions, with a novel and efficient ROB implementation. We demonstrate a significant performance improvement over the baseline in-order commit processor, achieving 95% of the performance of speculative out-of-order commit techniques with just a 4% overhead in power.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions and feedback to improve this work. This work was funded by a grant from the Singapore National Research Foundation (NRF2018NCR-NCR002), and a Startup Grant from the National University of Singapore.

REFERENCES

[1] 2010 (accessed August 21, 2020). *IEEE Floating-Point Arithmetic.* https://docs.oracle.com/cd/E19957-01/805-4940/6j4m1u7pj/index.html.
[2] Furat Afram, Hui Zeng, and Kanad Ghose. 2013. A group-commit mechanism for ROB-based processors implementing the x86 ISA. In *International Symposium on High Performance Computer Architecture (HPCA '13).* https://doi.org/10.1109/HPCA.2013.6522306
[3] Ismail Akturk and Ulya R. Karpuzcu. 2017. AMNESIAC: Amnesic automatic computer trading computation for communication for energy efficiency. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17).* https://doi.org/10.1145/3037697.3037741
[4] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott Baden, and Dean Tullsen. 2012. Redefining the role of the CPU in the era of CPU-GPU integration. *IEEE Micro* 32, 6 (2012), 4–16. https://doi.org/10.1109/MM.2012.57
[5] Gordon B Bell and Mikko H Lipasti. 2004. Deconstructing commit. In *International Symposium on Performance Analysis of Systems and Software (ISPASS '04).* https://doi.org/10.1109/ISPASS.2004.1291357
[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718
[7] Yuan Chou, Jason Fung, and John Paul Shen. 1999. Reducing branch misprediction penalties via dynamic control independence detection. In *International Conference on Supercomputing (ICS '99).* https://doi.org/10.1145/305138.305175
[8] Jamison D Collins, Dean M Tullsen, and Hong Wang. 2004. Control flow optimization via dynamic reconvergence prediction. In *International Symposium on Microarchitecture (MICRO '04).* https://doi.org/10.1109/MICRO.2004.13
[9] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-order commit processors. In *International Symposium on High-Performance Computer Architecture (HPCA '04).* https://doi.org/10.1109/HPCA.2004.10008
[10] Marc De Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *International Symposium on Microarchitecture (MICRO '11).* https://doi.org/10.1145/2155620.2155637

[11] Nam Duong and Alex V Veidenbaum. 2012. Compiler-assisted, selective out-of-order commit. *IEEE Computer Architecture Letters* 12, 1 (2012), 21–24. https://doi.org/10.1109/L-CA.2012.8

[12] Andrew Glew. 1998. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea Session* (Oct. 1998).

[13] Marius Grannaes, Magnus Jahre, and Lasse Natvig. 2011. Storage efficient hardware prefetching using delta-correlating prediction tables. *Journal of Instruction-Level Parallelism* 13 (2011), 1–16.

[14] Part Guide. 2011. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: System Programming Guide, Part* 2 (2011), 5.

[15] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *International workshop on workload characterization (WWC '01)*. https://doi.org/10.1109/WWC.2001.990739

[16] Tae Jun Ham, Juan L Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *International Symposium on Microarchitecture (MICRO '15)*. 191–203. https://doi.org/10.1145/2830772.2830800

[17] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[18] Milad Hashemi, Onur Mutlu, and Yale N Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *International Symposium on Microarchitecture (MICRO '16)*. https://doi.org/10.1109/MICRO.2016.7783764

[19] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[20] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *International Conference on Supercomputing (ICS '09)*. https://doi.org/10.1145/1542275.1542349

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO '04)*. https://doi.org/10.1109/CGO.2004.1281665

[22] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2013. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 1 (2013), 5. https://doi.org/10.1145/2445572.2445577

[23] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *International conference on Architectural support for programming languages and operating systems (ASPLOS '96)*. https://doi.org/10.1145/248208.237173

[24] Salvador Petit Marti, Julio Sahuquillo Borras, Pedro Lopez Rodriguez, Rafael Ubal Tena, and Jose Duato Marin. 2009. A complexity-effective out-of-order retirement microarchitecture. *IEEE Trans. Comput.* 58, 12 (2009), 1626–1639. https://doi.org/10.1109/TC.2009.95

[25] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *International Symposium on Microarchitecture (MICRO '02)*. https://doi.org/10.1109/MICRO.2002.1176234

[26] Daniel S McFarlin and Craig Zilles. 2015. Branch vanguard: Decomposing branch functionality into prediction and resolution instructions. In *International Symposium on Computer Architecture (ISCA '15)*. https://doi.org/10.1145/2749469.2750400

[27] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *International Symposium on High-Performance Computer Architecture (HPCA '09)*. https://doi.org/10.1109/HPCA.2003.1183532

[28] Vlad Petric, Anne Bracy, and Amir Roth. 2002. Three extensions to register integration. In *International Symposium on Microarchitecture (MICRO '02)*. 37–47. https://doi.org/10.1109/MICRO.2002.1176237

[29] Eric Rotenberg and Jim Smith. 1999. Control independence in trace processors. In *International Symposium on Microarchitecture (MICRO '99)*. https://doi.org/10.1109/MICRO.1999.809438

[30] Benjamin T Sander, William A Hughes, Sridhar P Subramanian, and Teik-Chung Tan. 2003. Stride based prefetcher with confidence counter and dynamic prefetch-ahead mechanism. US Patent 6,571,318.

[31] Niranjan Soundararajan, Saurabh Gupta, Ragavendra Natarajan, Jared Stark, Rahul Pal, Franck Sala, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. 2019. Towards the adoption of local branch predictors in modern out-of-order superscalar processors. In *International Symposium on Microarchitecture (MICRO '19)*. https://doi.org/10.1145/3352460.3358315

[32] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The RISC-V instruction set manual, volume I: Base user-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).